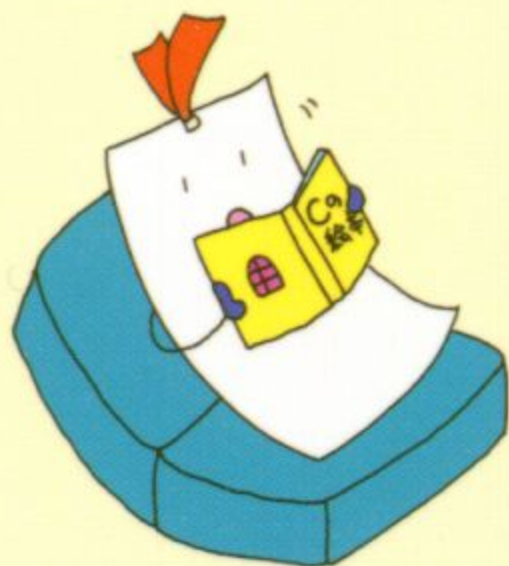
C言語が好きになる9つの扉

Cの絵本

（株）アンク 著

SE
SHOEISHA

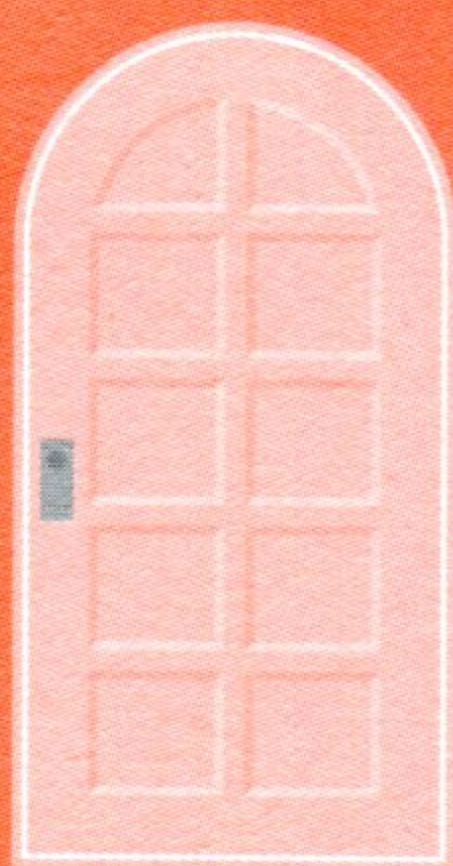
本書の特徴

C言語には
難解なトピックもあるため、
文章だけではなかなか
イメージがつかめず、
理解しづらいものですね。

本書はイラストで
解説してありますので、
直感的にイメージが
とらえられ、理解も
進んでいきます。

さあ、C言語への扉を開き、
プログラマーへの道を
進んでいきましょう!

C言語が好きになる9つの扉

Cの絵本

(株)アンク 著

SE
SHOEISHA

本書内容に関するお問い合わせについて

このたびは翔泳社の書籍をお買い上げいただき、誠にありがとうございます。弊社では、読者の皆様からのお問い合わせに適切に対応させていただくため、以下のガイドラインへのご協力をお願いしております。下記項目をお読みいただき、手順に従ってお問い合わせください。

●お問い合わせの前に

弊社Webサイトの「正誤表」や「出版物Q&A」をご確認ください。これまでに判明した正誤や追加情報、過去のお問い合わせへの回答（FAQ）、的確なお問い合わせ方法などが掲載されています。

正誤表　http://www.seshop.com/book/errata/
出版物Q&A　http://www.seshop.com/book/qa/

●ご質問方法

弊社Webサイトの書籍専用質問フォーム（http://www.seshop.com/book/qa/）をご利用ください（お電話や電子メールによるお問い合わせについては、原則としてお受けしておりません）。

※質問専用シートのお取り寄せについて

Webサイトにアクセスする手段をお持ちでない方は、ご氏名、ご送付先（ご住所／郵便番号／電話番号またはFAX番号／電子メールアドレス）および「質問専用シート送付希望」と明記のうえ、電子メール（qaform@shoeisha.com）、FAX、郵便（80円切手同封）のいずれかにて“編集部読者サポート係”までお申し込みください。お申し込まれた手段にそって、折り返し質問シートをお送りいたします。シートに必要事項を漏れなく記入し、“編集部読者サポート係”までFAXまたは郵便にてご返送ください。

●回答について

回答は、ご質問いただいた方法にそってご送付いたします。ご質問の内容によっては、回答に数日ないしはそれ以上の期間を要する場合があります。

●ご質問に際してのご注意

本書の対象を越えるもの、記述個所を特定されていないもの、また読者固有の環境に起因するご質問等にはお答えできませんので、あらかじめご了承ください。

●郵便物送付先およびFAX番号

送付先住所　〒160-0006　東京都新宿区舟町5
FAX番号　03-5362-3818
宛先　（株）翔泳社 出版局 編集部読者サポート係

はじめに

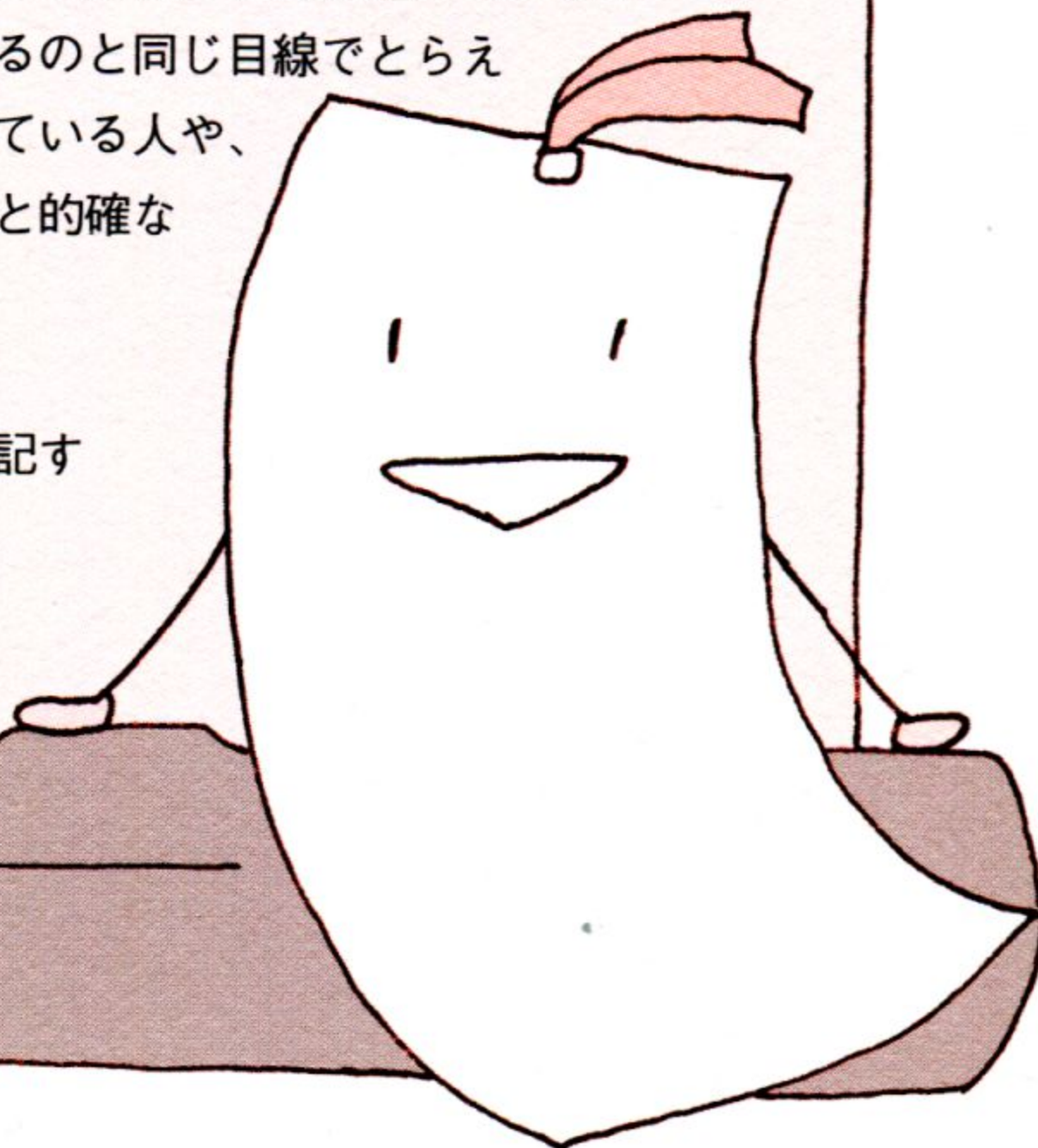
本書はC言語の入門書です。C言語は、数あるプログラミング言語の中でも、世界代表選手といってもよいほど普及している言語です。でも、実をいうとC言語は決して易しい言語ではありません。この本を手に取った方の中には、一度C言語にチャレンジしたものの、難しそうでやめてしまったという方もいるかもしれません。

さて、それではプログラムを作ることの難しさっていったい何なのでしょう？ ひとつは、「プログラムの組み立て」があります。つまり、ある機能を実現するために、どのようにプログラミング言語のコードを組み合わせていけばよいかということです。これは、経験があるほど有利ですが、人によって向き・不向きがあり、ある程度のセンスも必要です。

でも、多くの人はもう少し手前の、「言語仕様の理解」のところでつっかえてしまいます。そもそも、プログラミング言語は、それを作った人が試行錯誤して、「こうしたら便利じゃないかな」と考えた結果生まれてきたものです。そういう概念的な部分は、わかる人にとってはあたり前のことですが、知らない人にとっては理解するのがとても難しく、いったい何の役にたつのか不思議なものです。

実際、私が初心者にプログラミングを教えるときも、イメージがなかなか伝わらなくて苦労することがよくあります。プログラミングというと、論理的な作業と思われがちですが、実は想像力も必要な作業なのです。特にC言語の場合、ポインタなど初心者には難解なトピックもあり、それに至る過程でしっかりしたイメージを養っておかないと、すぐに行き詰まってしまいます。

そこで、この本では、C言語やプログラミングの基礎を知らない初心者向けに、文章よりもイメージを前面に出した解説をしています。プログラムの流れや、難解といわれるポインタやメモリの構造を、プログラマが実際にとらえているのと同じ目線でとらえることができます。他の本を読んでもわからなくて苦労している人や、理解が中途半端でいまいちピンときていない人にも、きっと的確なイメージが伝わると思います。

2002年 2月　著者記す

≫本書の特徴

- 本編は見開き2ページで1つの話題を完結させ、イメージがバラバラにならないように配慮しています。また、あとで必要な部分を探すのにも有効にお使いいただけます。
- もちろん、概念論だけでおわらないように、サンプルプログラムも豊富に用意しました。見開きの中で紹介する小さなプログラムの他に、章末に比較的長く、実用的なサンプルプログラムがあります。なお、本書の解説とサンプルの多くは、Microsoft Windows 2000上で、Visual C++6.0を使って開発することを前提にしましたが、UNIX上のgccなど、ANSI Cをサポートした C言語開発環境であれば、そのまま動作させることができます。
- 少し高度なトピックスについては、付録として掲載しましたので、本編が理解できたと思う方は読み進めてみてください。また、付録には実際にプログラムの開発を行う際の手引きとなる情報や資料も載せています。

≫対象読者

本書はC言語をこれから学ぶ方はもちろん、一度はじめたけれども挫折してしまったという方にもお勧めします。また、プログラミングの基礎的な事項も説明してありますので、プログラミング初心者の方にもきっとお役にたちます。

≫表記について

本書は以下のような約束で書かれています。

【例と実行結果】

プログラミングで入力する内容

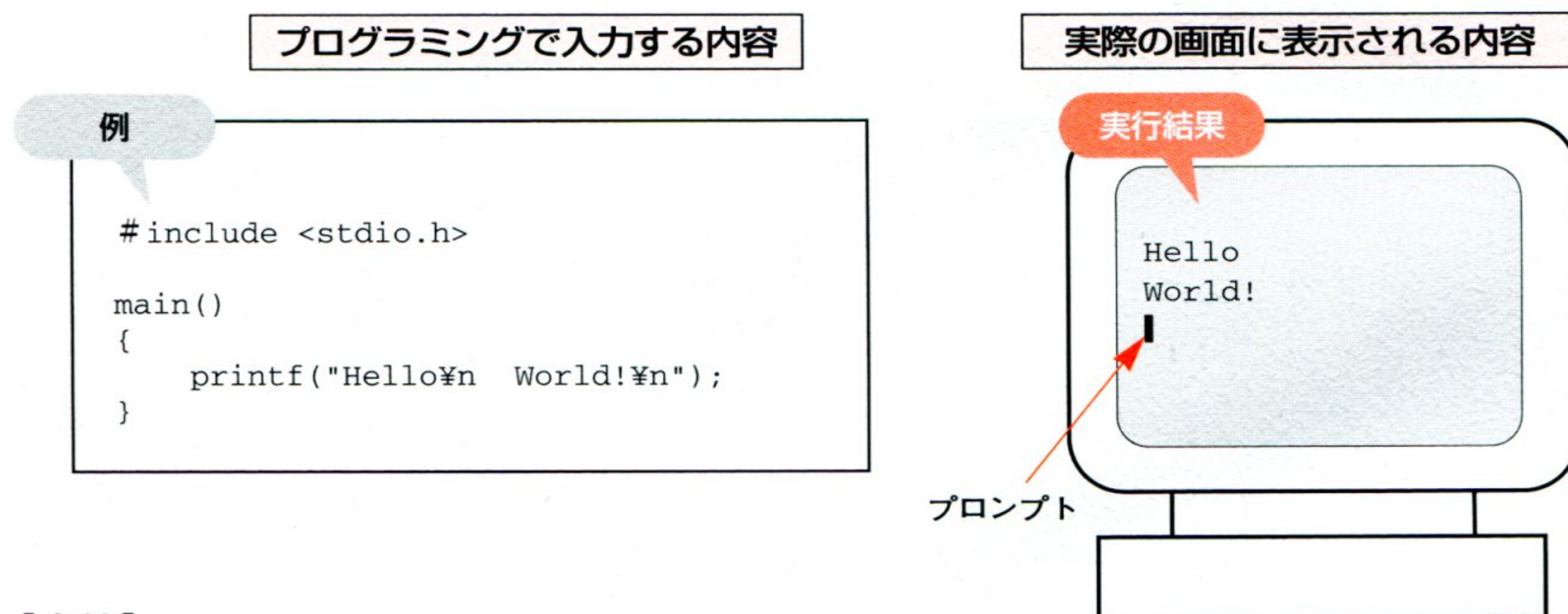
例

```
#include <stdio.h>

main()
{
    printf("Hello¥n  World!¥n");
}
```

実際の画面に表示される内容

実行結果

```
Hello
World!
▌
```

プロンプト

【書体】

ゴシック体：重要な単語

`List Font`：C言語のプログラミングに実際に用いられる文や単語

`List Bold Font`：`List Font`の中でも重要なポイント

【その他】

- 本文中の関数などに読み方のルビを振ってありますが、あくまで一例であり、異なる読み方をする場合もあります。
- 必要なヘッダファイルはVisual C++6.0の場合です。Visual C++6.0以外の場合はコンパイラのマニュアルで確認してください。

もくじ

お買い得

C言語をはじめる前に

C言語の位置づけ

コンピュータで動くプログラムを作成・記述するための言葉を「プログラミング言語」といいます。C言語もプログラミング言語の1つで、その代表格といってもよいほど普及しています。

C言語は1972年頃にUNIX(ユニックス)システム開発のために考え出されました。コンピュータ言語の中でも比較的古い言語といえるでしょう。その優れた言語仕様により、後に大型のコンピュータからパソコンに至るまで、活躍の場を広げました。

コンピュータの普及とともに、BASIC(ベーシック)、Java(ジャバ)、Perl(パール)など、よりわかりやすく、便利なプログラミング言語が登場してきましたが、現在でも市販のソフトウェアの多くがC言語で開発されるなど、C言語は「最強のプログラミング言語」としての勢力を保っています。それには次のような理由があります。

細かいレベルまでプログラムで制御が可能

BASICなどは細かい点に気を使わなくても、プログラムが簡単に書ける反面、言語にフォローしてもらった部分に手を出しにくいという側面もあります。C言語なら、比較的システムの深いところまで、取り扱えます。

移植性が高い

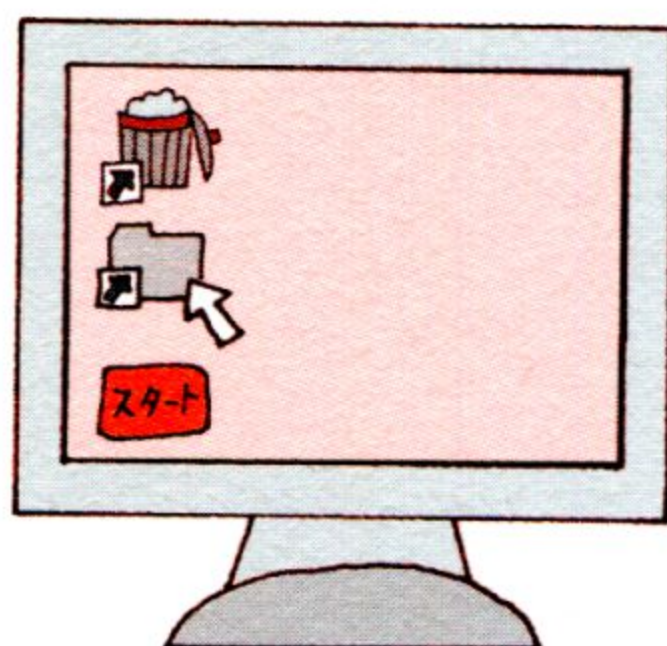
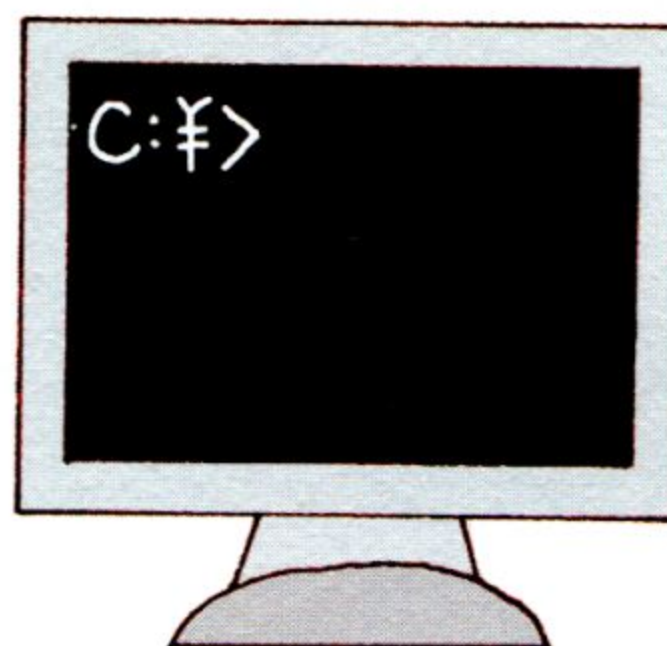
ANSI(アンシー)が定めた標準的な仕様があり、ソースレベルでの互換性が高いので、UNIXとWindowsなど異なるOS間でもソースコードのやり取りが容易にできます。よく普及しているため、ノウハウも豊富です。

記述が容易

他のプログラミング言語と比べて、記述方法が柔軟かつ記号的で、はじめて見た人にとっては少しわかりづらい面があるものの、一度理解してしまうと簡潔で記述しやすいものだとわかります。

C言語を学ぶのは楽ではありませんが、C言語を元に作られたプログラミング言語も多く、身につけておけば、きっと他のプログラミング言語にも応用ができるでしょう。

C言語の動作環境

C言語のプログラムは、基本的にMS-DOS（エムエスドス）やUNIXなどのCUI（キャラクターユーザーインターフェイス）の環境で動きます。WindowsなどのGUI（グラフィカルユーザーインターフェイス）の環境では、コマンドプロンプト（DOSプロンプト）を起動し、その中で実行します。

コマンドプロンプトでプログラムを実行

```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:¥>cd program

C:¥program>prtarg languageC BASIC Java HTML
1・・・languageC
2・・・BASIC
3・・・Java
4・・・HTML

C:¥program>
```

CUIの環境は、GUIに比べると見た目に楽しいものではありません。これから作っていくプログラムを見ると、市販のゲームソフトや表計算ソフトなどのアプリケーションには程遠いと思うかもしれません。しかし、これらのソフトの多くがC言語と専用の開発キットを利用して作られているのです。

また、CGI（Webサーバ上で動くアプリケーション）の開発にはPerlを使うのが主流となっていますが、処理速度が重要なときはC言語を利用することがあります。

プログラミングから実行までの流れ

プログラミングを行うには、ただコンピュータさえあればよいというわけではありません。まず、C言語を記述するための「**テキストエディタ**（Windowsでいうメモ帳など）」が必要です。そして、このCのソースプログラムをコンピュータがわかる言葉（機械語）に変換する、C言語の「**コンパイラ**」が必要です。エディタとコンパイラがセットになったソフト（Microsoft Visual C++など）も市販されています（詳細は第8章および付録参照）。

1 ".c"という拡張子のテキストファイルにC言語のプログラムを記述する。

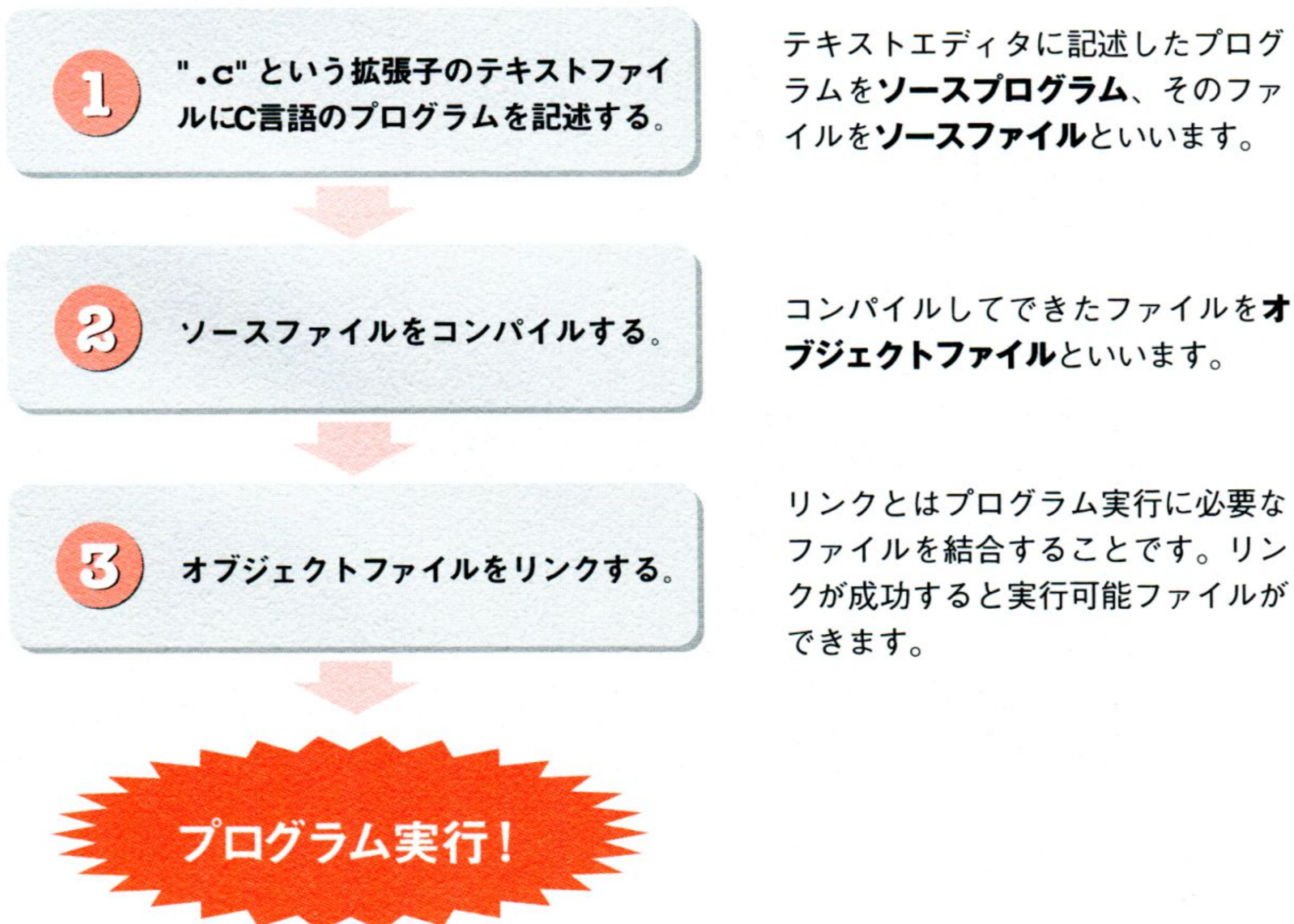
テキストエディタに記述したプログラムを**ソースプログラム**、そのファイルを**ソースファイル**といいます。

2 ソースファイルをコンパイルする。

コンパイルしてできたファイルを**オブジェクトファイル**といいます。

3 オブジェクトファイルをリンクする。

リンクとはプログラム実行に必要なファイルを結合することです。リンクが成功すると実行可能ファイルができます。

プログラム実行！

プログラム記述時の約束

正常に動くプログラムを作るには、次の約束を守って記述してください。

①原則として半角で記述する

日本語対応のコンパイラを使用した場合、コメントおよび ""（ダブルクォーテーション）内は全角記述が可能です。

②半角カナは使わない

"" の中でも使用しないことをお勧めします。

③全角スペースの使用に注意する

"" の外に書くとエラーになります。発見しにくいので要注意です。

④小文字と大文字を区別して書く

たとえばifとIFはまったく別のものです。

⑤コメントは/*と*/でくくる

プログラムに反映したくない説明的な記述を/*　*/の中に書くことができます。

⑥予約語に気をつける

予約語はコンパイラが使用するキーワードです。
それぞれの持つ働き以外の目的で使用することはできません。

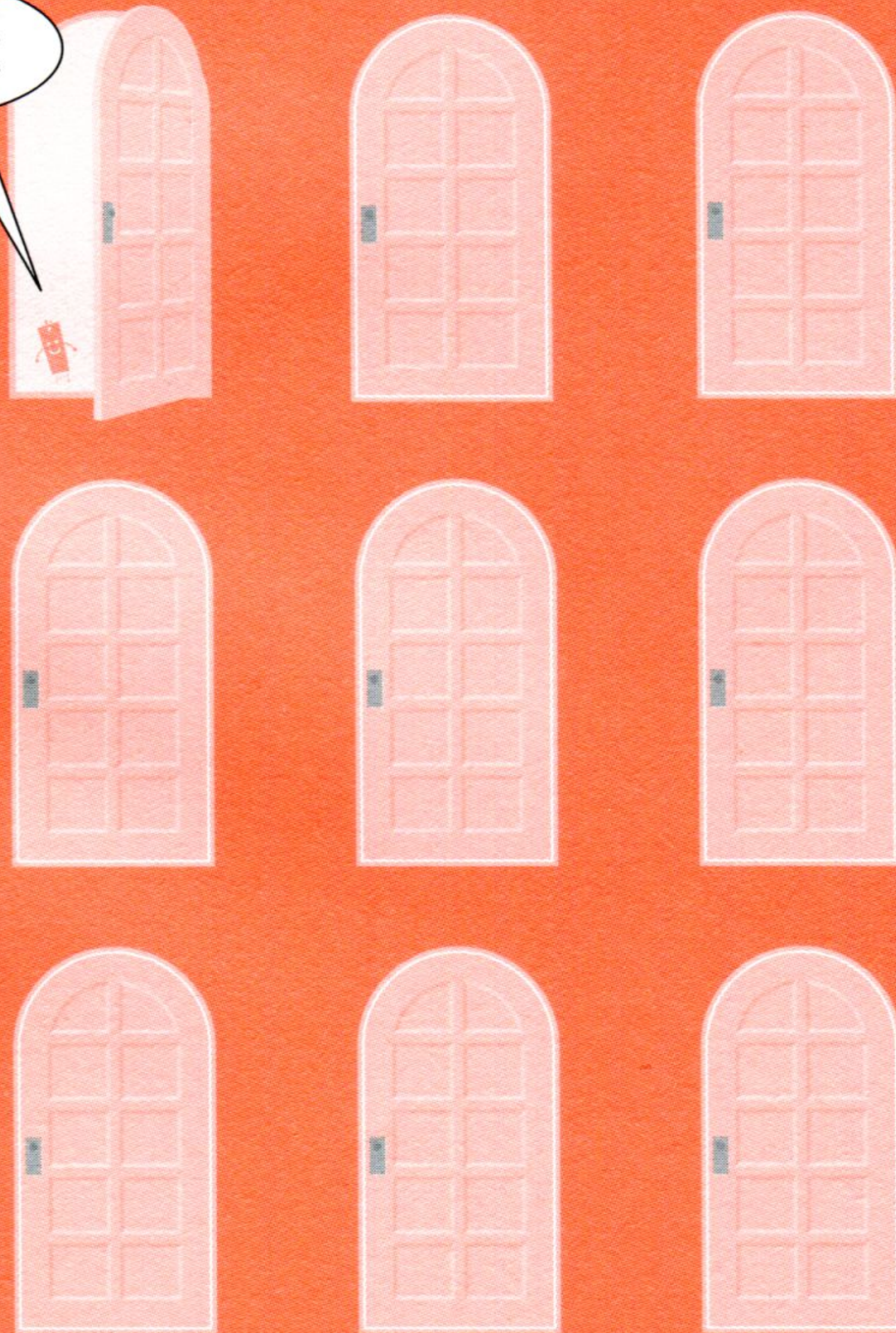
予約語一覧

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

1 基本的なプログラム

第1章

第1章は

ここがkey

まずは文字の表示から

これからいよいよ、実際にプログラムを作っていきます。まず、画面に「Hello World!」と表示させることからはじめましょう。表示させる文字列は何でもよいのですが、こういうときは昔から「Hello World」とすることが多いようです。プログラムをはじめるのにふさわしい言葉だからでしょうか。

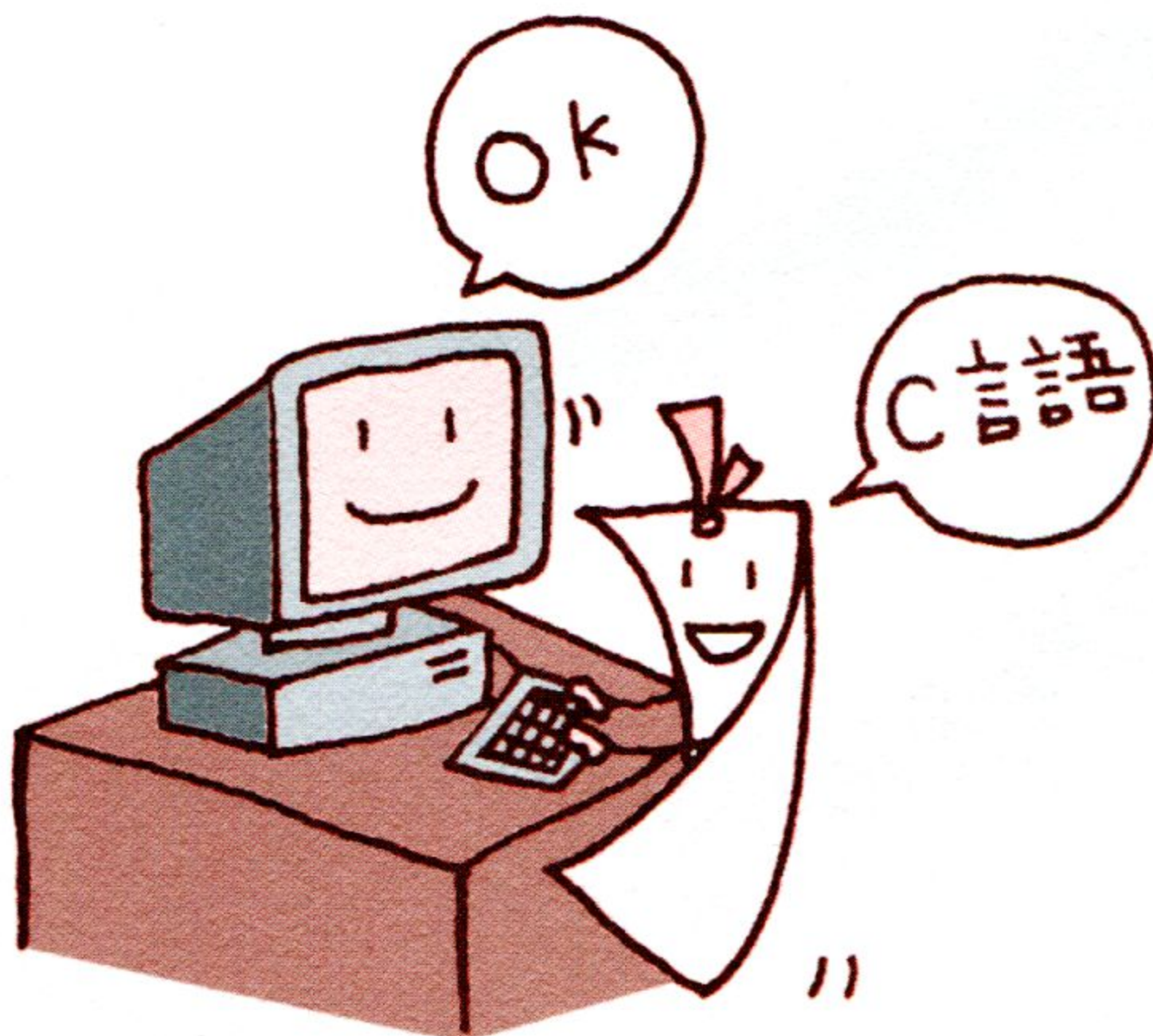
C言語で文字を表示するときには、**printf()**（プリントエフ）という**関数**を使います。このように最後に()をつけて書くときは、それが関数であることを表します。関数というと、数学で苦労して拒否反応を示す人もいるかもしれませんね。ただし、C言語でいうところの関数とは「一連の処理の集まり」であり、数学のそれよりもう少し広い意味を持っています。

そして、これらの関数などの処理を**main()**（メイン）という関数の中に書いていきます。main()関数はプログラムの開始地点（エントリポイント）であり、コマンドラインなどからプログラムを起動すると、main()関数の処理が最初に実行されます。

このように、C言語は関数の集まりでできています。関数についての詳細は第5章で解説することにしましょう。

OK

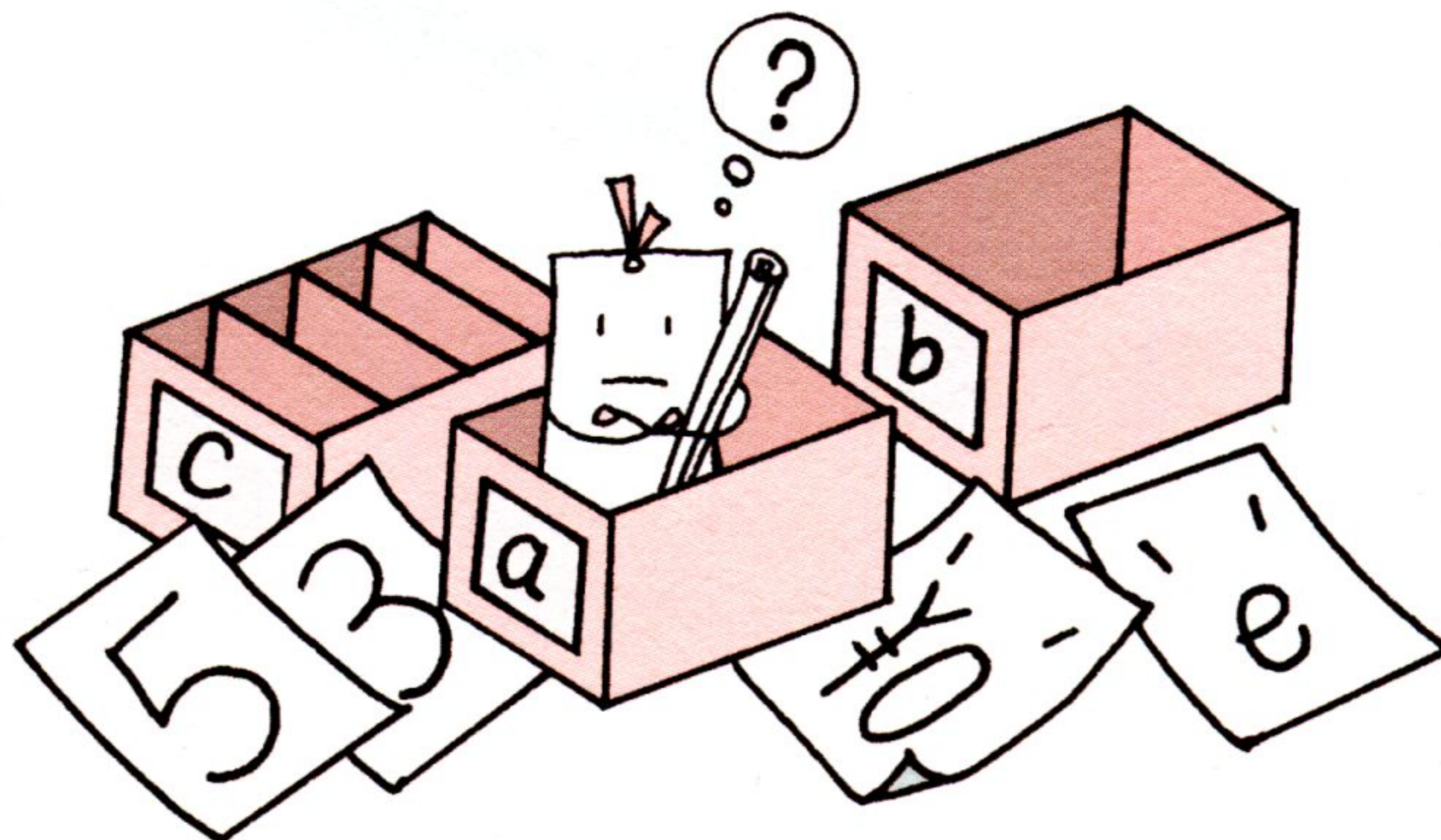
C言語

Topics

C いろいろな型、値、変数

ただ決まったメッセージを表示するだけではつまらないので、次のステップとして、計算の結果を表示してみます。`printf()`には書式を指定して値を表示する機能があるのです。この機能を用いて、いろいろなタイプの**定数**（**値**）を表示します。

そして、それらを格納しておくための箱である**変数**についても勉強していきます。C言語など多くのプログラミング言語では、その使い道によって、**整数型**、**実数型**、**文字型**などの**型**（かた）を定めています。そして、さらにその中でも精度などによって型を分けています。このように厳密に型を分けるのは、コンピュータが「こっちは整数、あっちは文字列」などと柔軟に判断するのが苦手であり、コンピュータの搭載メモリが有限であるからです。最近ではメモリの搭載量も飛躍的に向上していますが、それでも歯止めをかけていかないとすぐに足りなくなってしまいます。

C言語で変数と値について述べるとき、**文字**と**文字列**は少しやっかいです。後半では、文字と文字列の関係、文字と**ASCII**（アスキー）**コード**の関係、**制御コード**などについて解説していきます。

それでは、次のページからC言語プログラミングのはじまりです。

Hello World!

最初に、プログラムの基本的な書き方や、画面への文字列の表示方法について見ていきましょう。

プログラムを作る

一番簡単なC言語のプログラムは次のようなものです。このプログラムを実行すると、「Hello」と「World!」という文字列を画面に表示します。

例

```
#include <stdio.h>

main()
{
    printf("Hello¥nWorld!¥n");
}
```

printf( )を使うために必要です。

文字列を表示します。

実行結果

```
Hello
World!
```

»プログラムの基本形

C言語のプログラムの基本形は次のようになります。

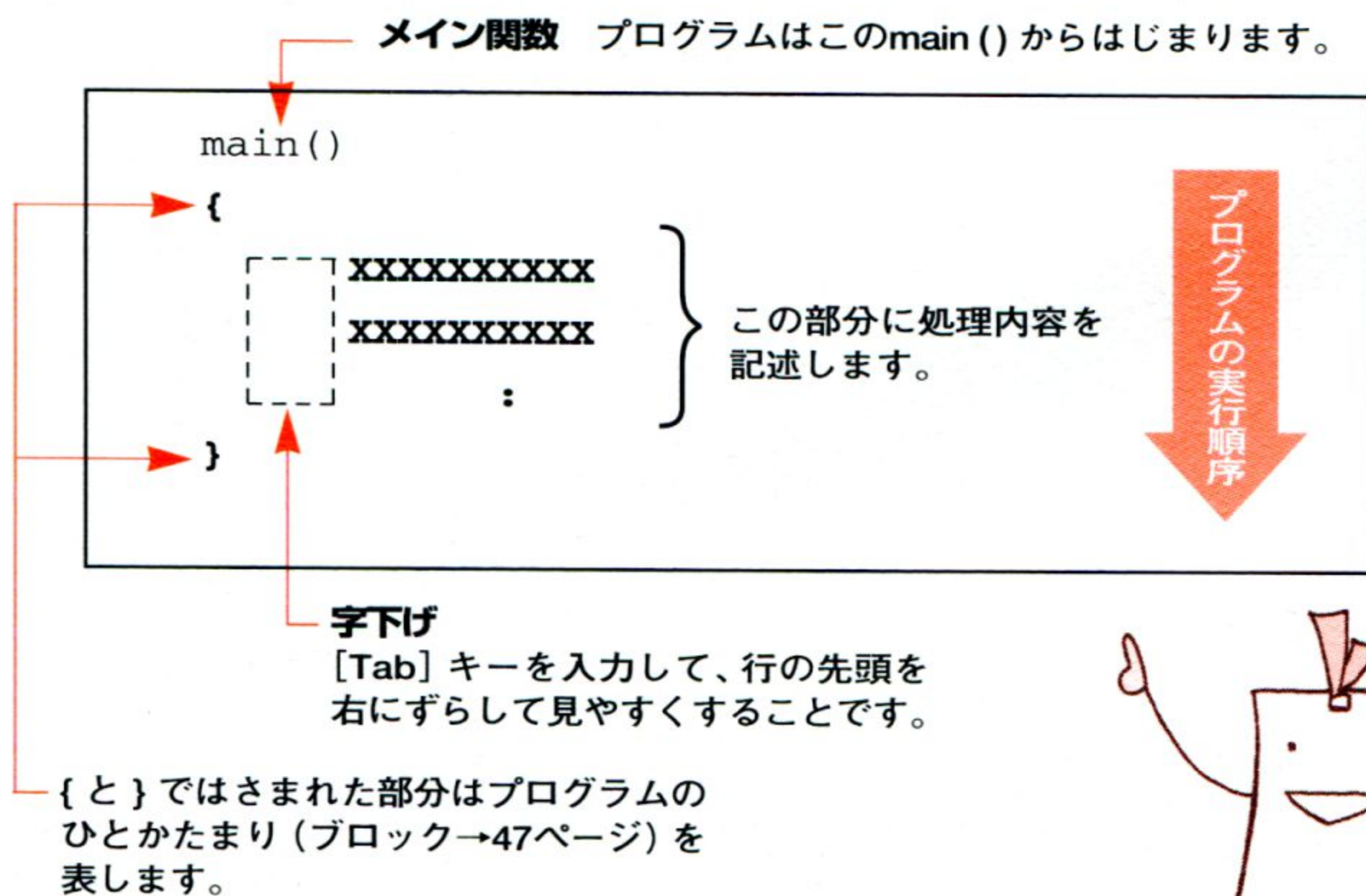
```
main()
{
    XXXXXXXXXX
    XXXXXXXXXX
        :
}
```

メイン関数　プログラムはこのmain ( ) からはじまります。

この部分に処理内容を記述します。

プログラムの実行順序

字下げ
[Tab] キーを入力して、行の先頭を右にずらして見やすくすることです。

{ と } ではさまれた部分はプログラムのひとかたまり (ブロック→47ページ) を表します。

main( )がないと、コンパイル・実行ができません。

C 文字列を表示する

C言語のプログラムで文字列を表示するには、printf()関数を使います。

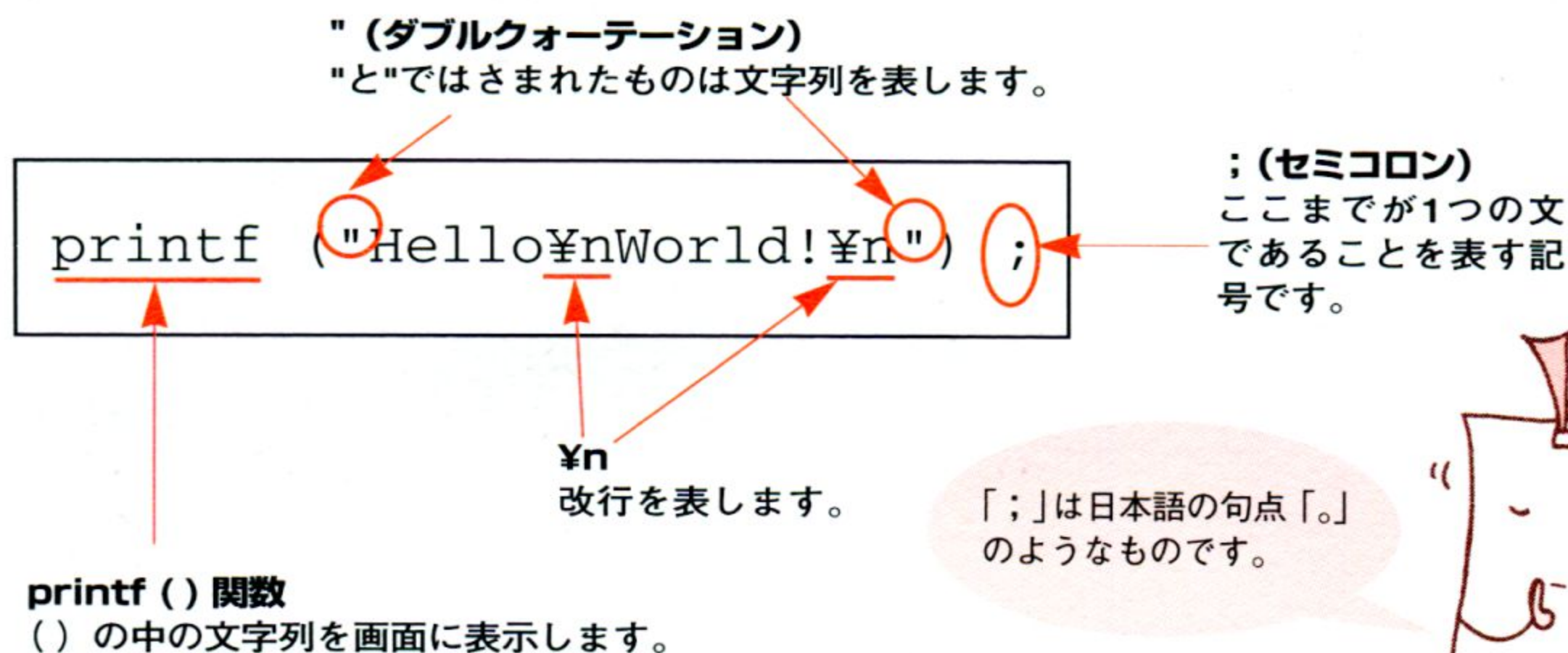
```
printf ("Hello¥nWorld!¥n") ;
```

"（ダブルクォーテーション）
"と"ではさまれたものは文字列を表します。

;（セミコロン）
ここまでが1つの文であることを表す記号です。

¥n
改行を表します。

printf()関数
()の中の文字列を画面に表示します。

「;」は日本語の句点「。」のようなものです。

≫¥nの役割

¥とその直後の1文字は特殊な文字の表示や操作を行います。たとえば¥nは改行（次行の先頭へ表示位置を移動する）を表します。この2文字は画面にそのまま表示されないことに注意してください。

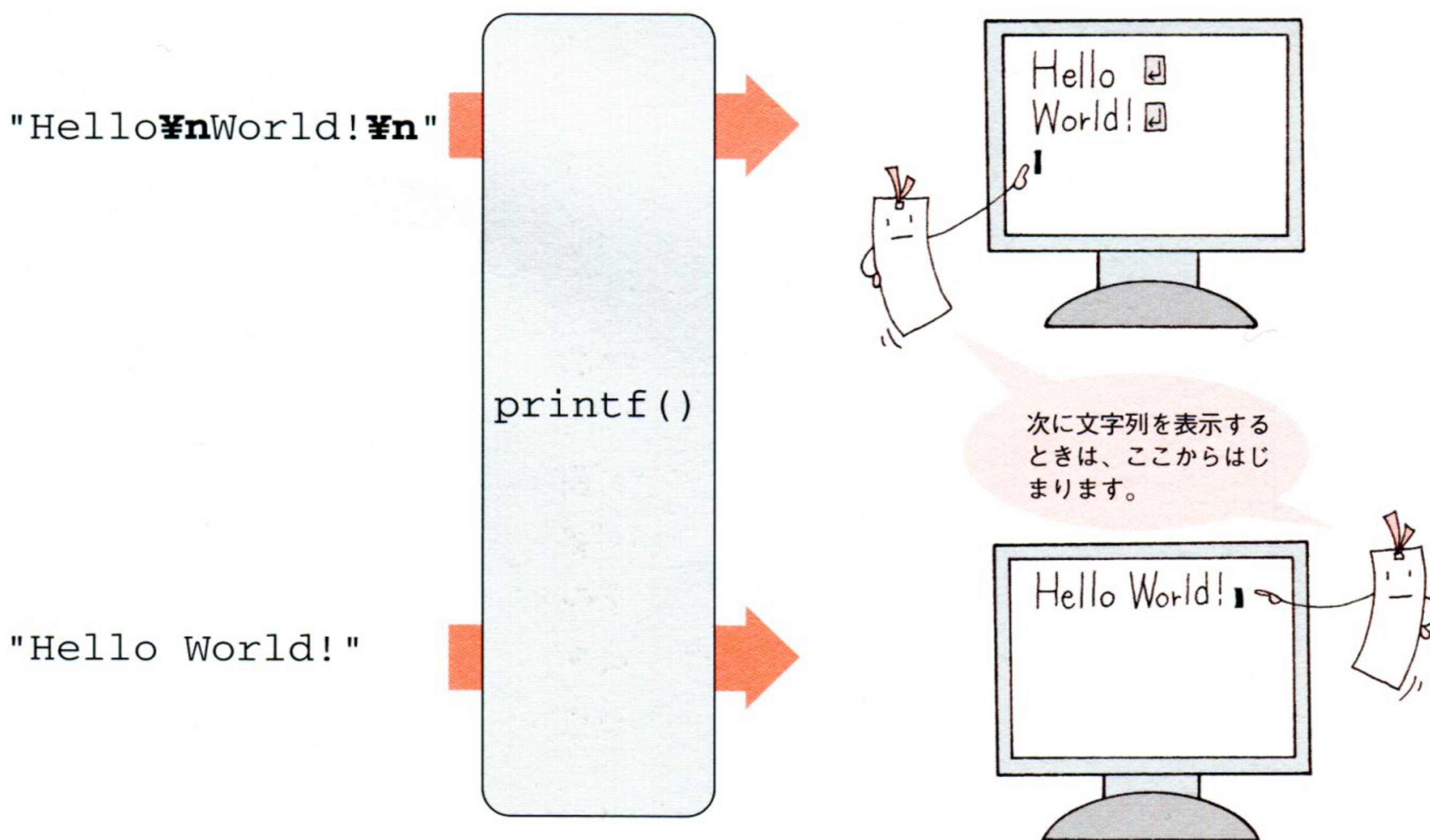
"Hello¥nWorld!¥n"

printf()

Hello
World!

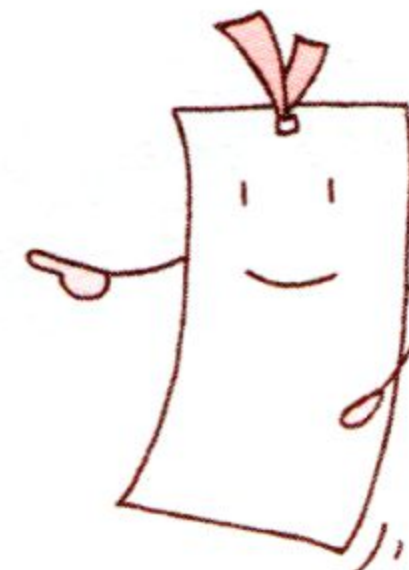
次に文字列を表示するときは、ここからはじまります。

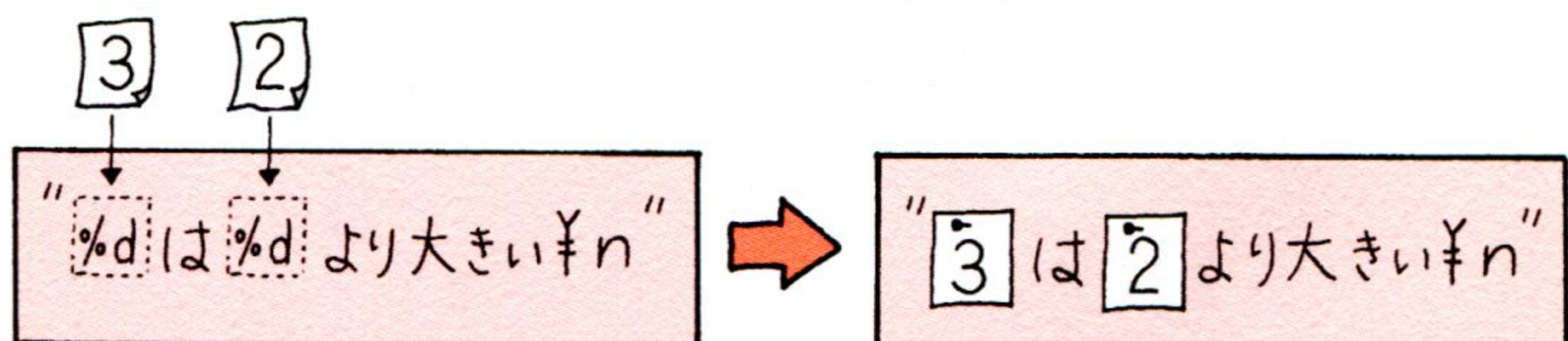
"Hello World!"

Hello World!

printf( )と定数

printf()を用いると文字列以外のデータも表示できます。その方法を見ていきましょう。

printf()の使い方

printf()には、ただ決まった文字列を表示するだけではなく書式を指定してデータを表示する機能があります。次の2つはどちらも画面に「3」を表示します。

複数のデータを表示するときの対応は次のようになります。

例

```
#include <stdio.h>

main()
{
    printf("%d-%dは%dです。¥n", 3, 2, 3-2);
}
```

実行結果

```
3-2は1です。
```

3と2と3-2の計算結果（=1）をそれぞれ整数として表示します。

»いろいろな書式指定

%dは整数を**10進数**（→34ページ）で表示する書式指定です。
書式指定は表示するデータの種類によって異なり、次のようなものがあります。

書式指定	意味	データの例
%d	整数（小数点のついていない数）を10進数で表示する	1、2、3、−45
%f	実数（小数点のついている数）を表示する	0.1、1.0、2.2
%c	文字（'で囲まれた半角文字1個）を表示する	'a'、'A'
%s	文字列（"で囲まれた文字）を表示する	"A"、"ABC"、"あ"

例

```
#include <stdio.h>

main()
{
    printf("%s %c %f ¥n", "6÷5", '=', 1.2);
}
```

実行結果

```
6÷5 = 1.200000
```

小数点の桁数（この例の場合0の数と考えてよい）は処理系により異なる場合があります。

書式とデータの種類は統一しなければいけません。

```
printf("%f", 2);
```

実数 × 整数

2.0としないと正しく表示できません。

変数

変数とは、数値や文字を格納しておく箱のようなものです。ここでは整数の値を変数に入れる方法を説明します。

宣言と代入

次のようにして変数を作成し、その中に値を入れることができます。

```
int a;
```

…整数（integer）の値が入るaという名前の変数を用意します。
これを「int型の変数aを**宣言**する」といいます。

変数
値を入れるための箱のようなものです。

変数名
変数名には半角英数字と「_（アンダースコア）」が使えます。ただし、先頭文字は数字ではいけません。

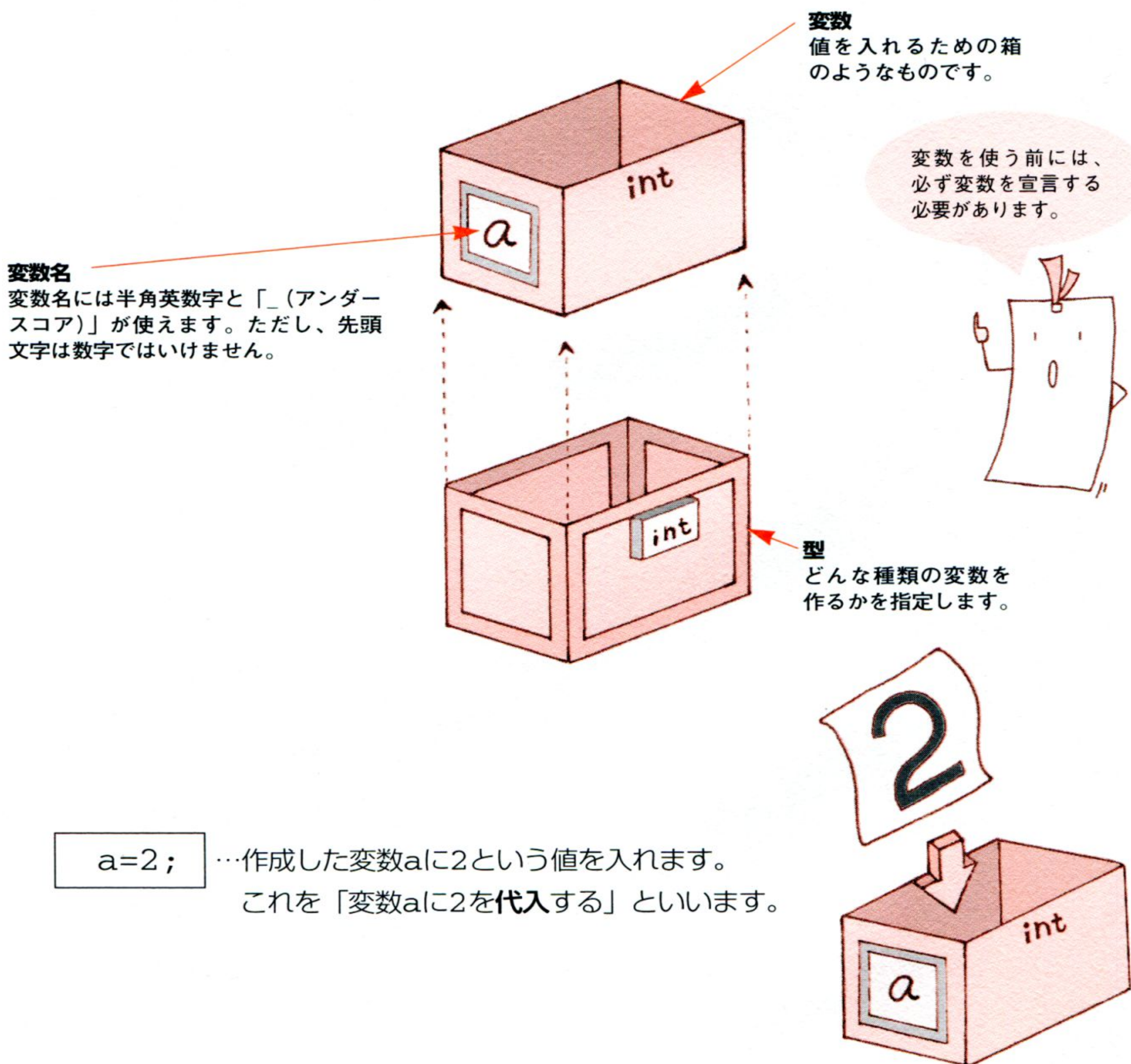
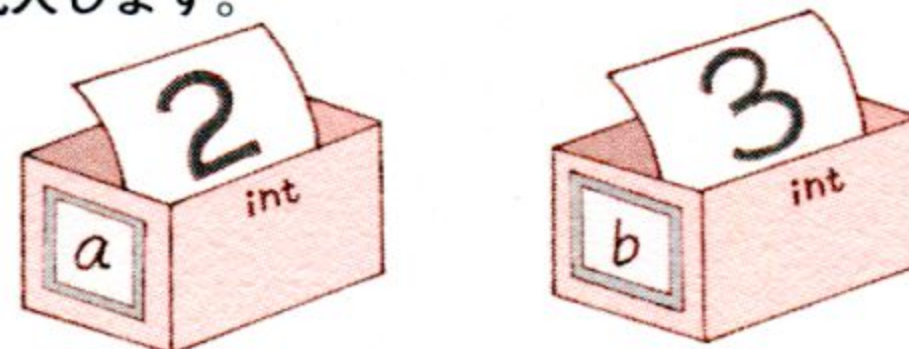
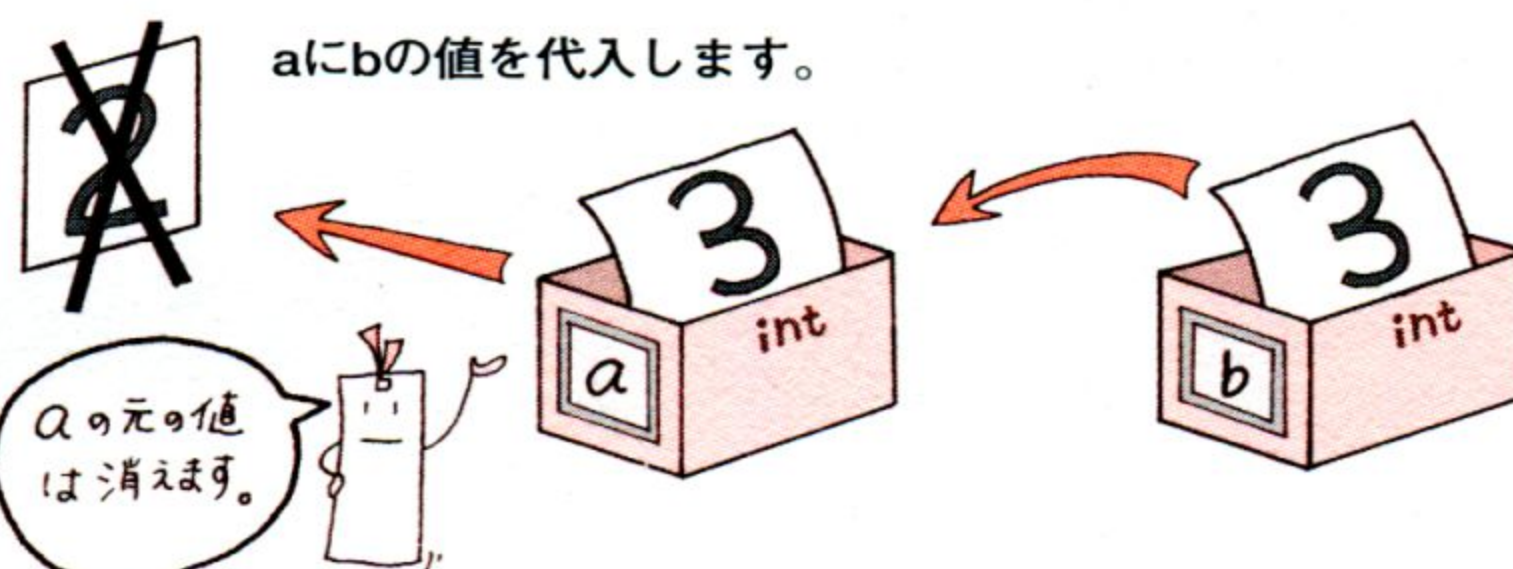
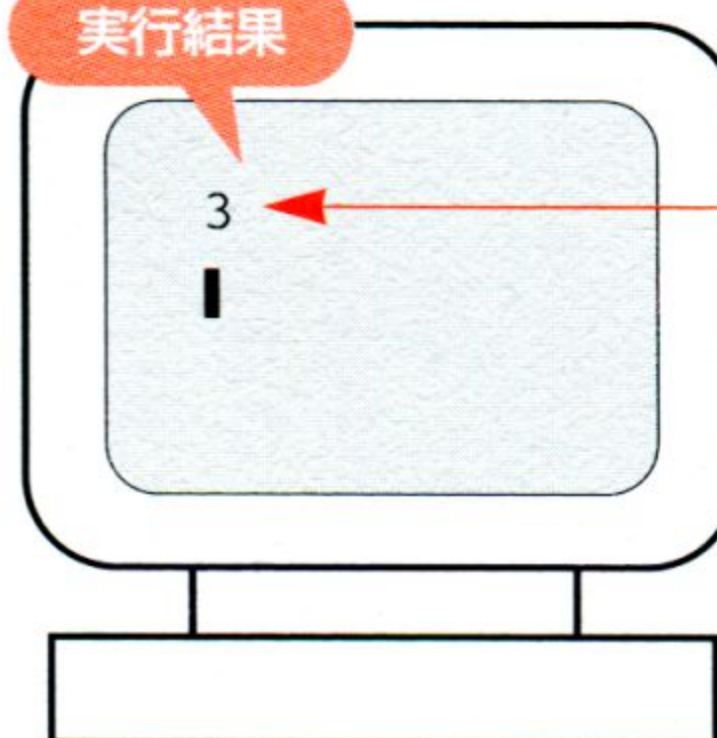
型
どんな種類の変数を作るかを指定します。

変数を使う前には、必ず変数を宣言する必要があります。

int
a
int
int
a
2

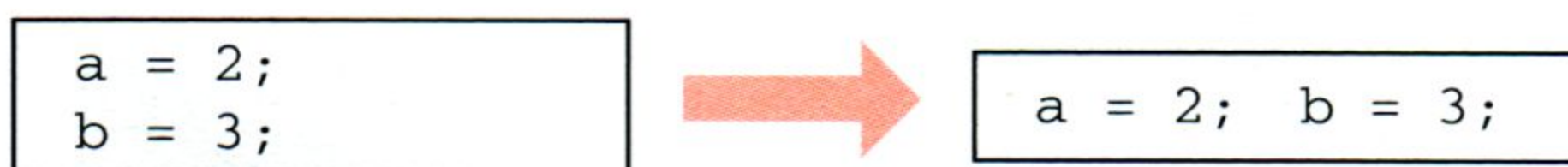
```
a=2;
```

…作成した変数aに2という値を入れます。
これを「変数aに2を**代入**する」といいます。

例

```
#include <stdio.h>

main()
{
    int a;
    int b;
    a = 2;
    b = 3;
    a = b;

    printf("%d¥n", a);
}
```


≫宣言の書き方

文は ; で区切りますが、１行に並べて書くことも可能です。

```
a = 2;
b = 3;
```
→
```
a = 2;  b = 3;
```

変数の宣言や値の代入を次のようにまとめることができます。

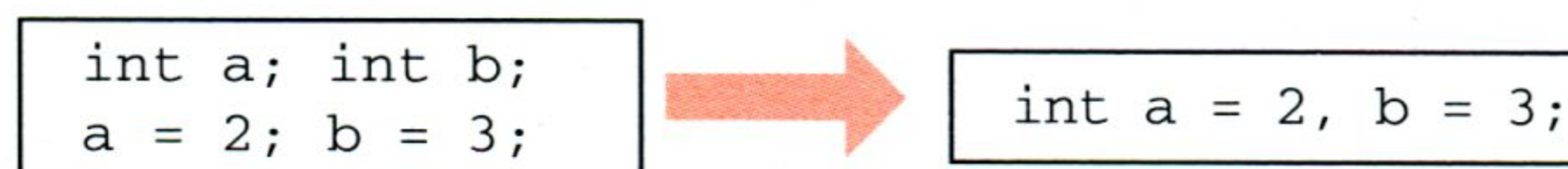
2変数の宣言

```
int a; int b;
```
→
```
int a, b;
```

1変数の宣言と代入

```
int a; a = 2;
```
→
```
int a = 2;
```

2変数の宣言と代入

```
int a; int b;
a = 2; b = 3;
```
→
```
int a = 2, b = 3;
```

宣言と代入を同時に行うことを「変数を**初期化**する」といいます。
初期化をすれば値を代入するのを忘れることはありませんし、プログラムも見やすくなります。

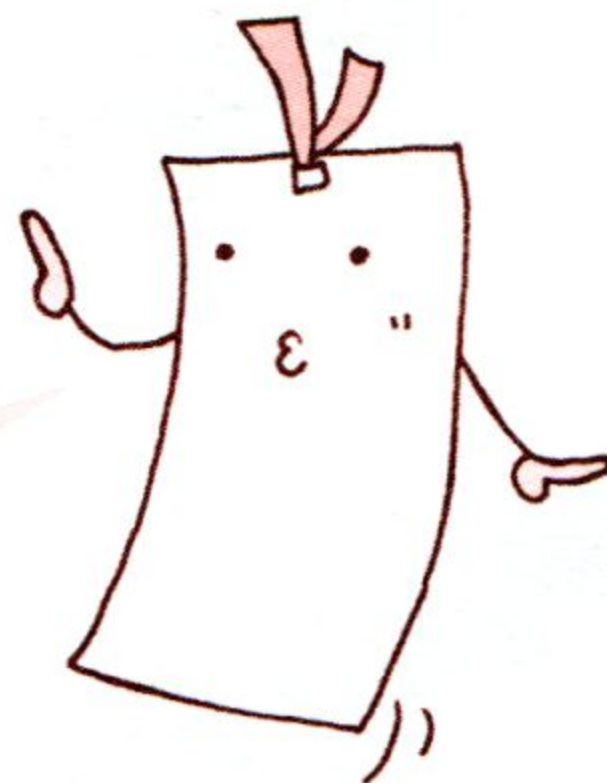
数値型

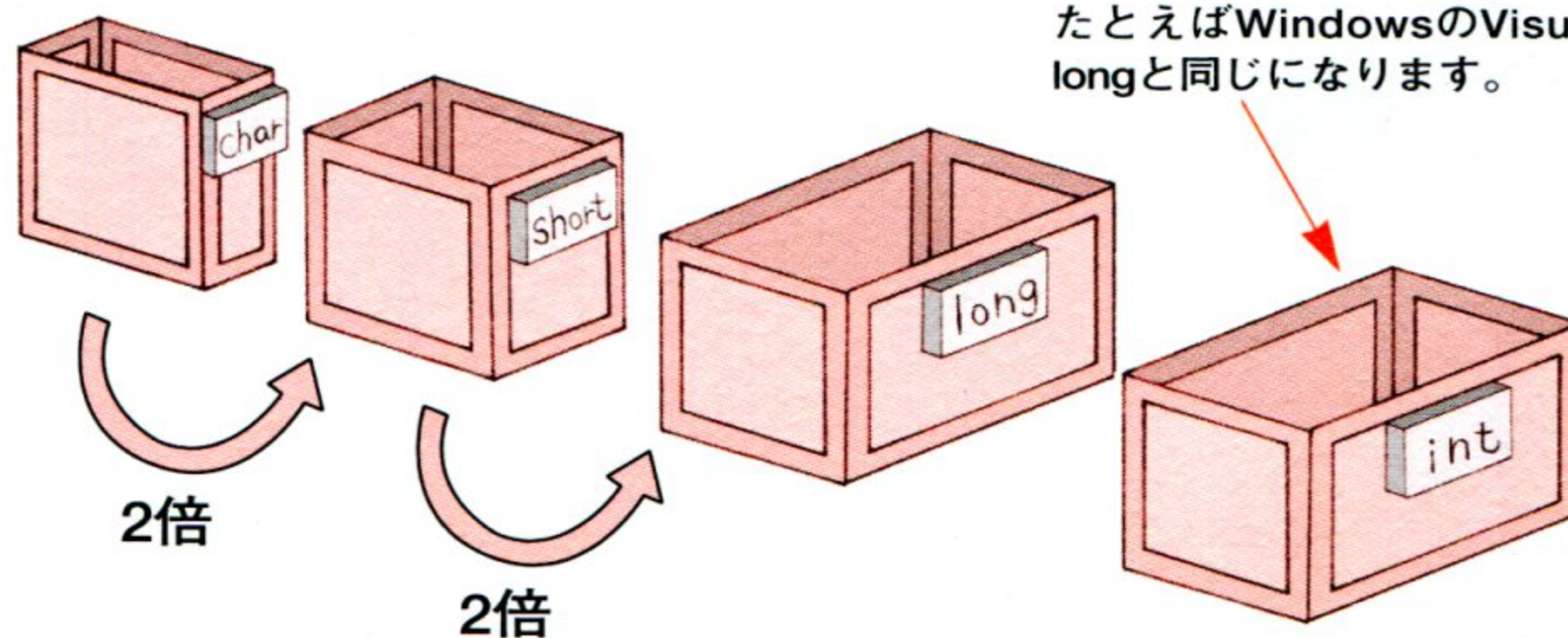
数値が入る変数の型には整数用の整数型と、実数用の実数型があります。

整数型

整数型には次のようなものがあります。

型の名前	読み方	入る値の範囲	サイズ(ビット数)
int	イント	システムにより異なる	-
unsigned int	アンサインド イント	システムにより異なる	-
long	ロング	−2147483648 〜2147483647	32
unsigned long	アンサインド ロング	0〜4294967295	32
short	ショート	−32768〜32767	16
unsigned short	アンサインド ショート	0〜65535	16
char	チャー、キャラ	−128〜127	8
unsigned char	アンサインド チャー、 アンサインド キャラ	0〜255	8

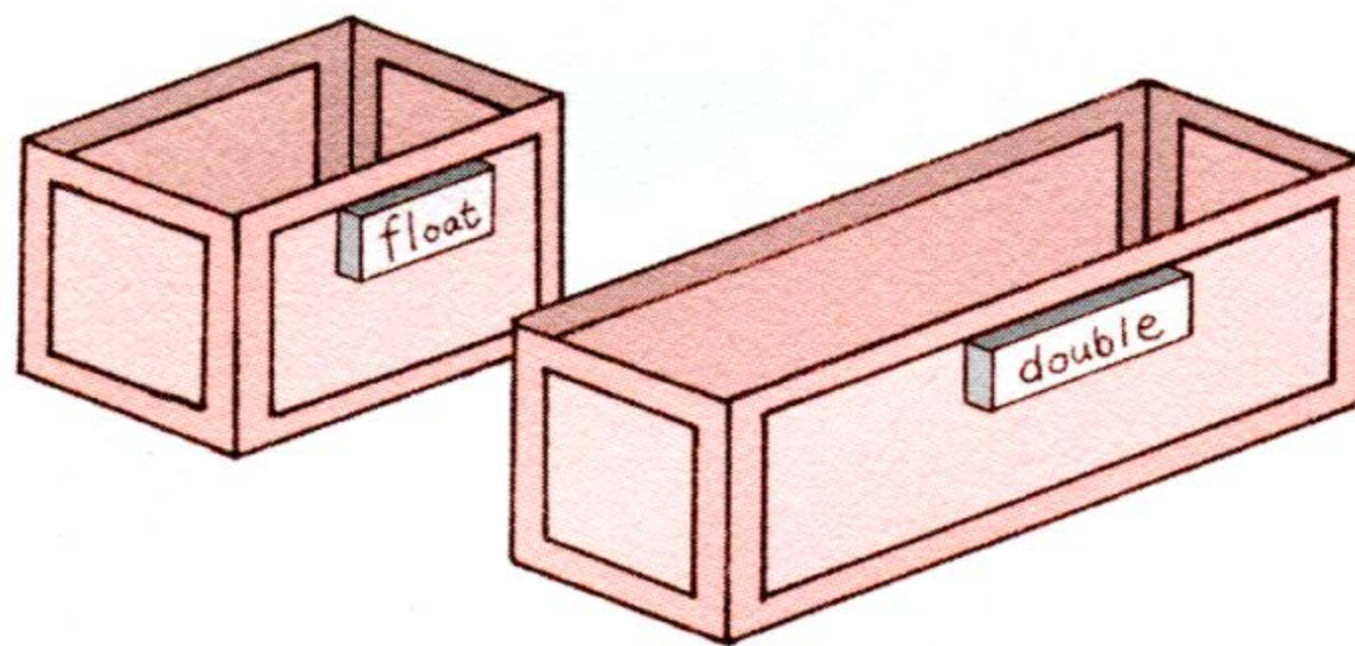
unsignedは
「符号がない」
という意味です。

型によってメモリを使う量は異なります。

intの範囲はそのシステムで処理の基本となる大きさになります。
たとえばWindowsのVisual C++では、longと同じになります。

C 実数型

実数型には次のようなものがあります。

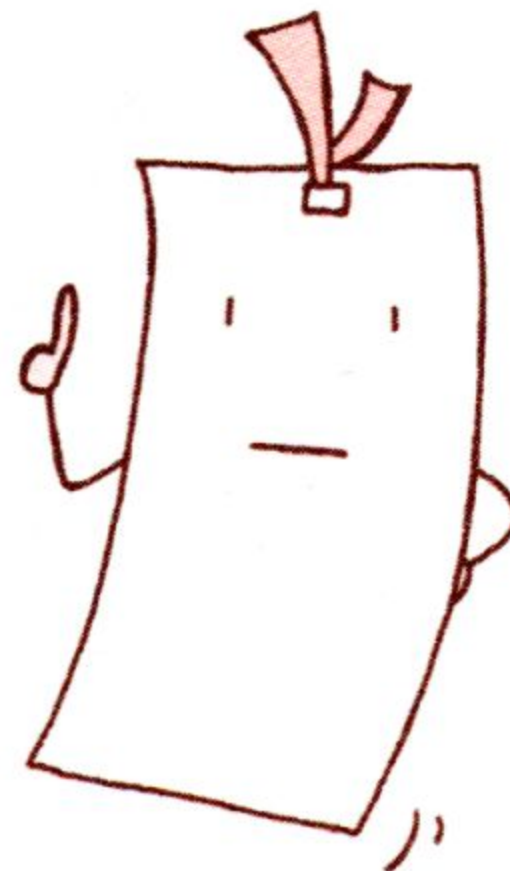
型の名前	読み方	入る値のおおまかな範囲	サイズ（ビット数）
float	フロート	$-3.4 \times 10^{38} \sim 3.4 \times 10^{38}$	32
double	ダブル	$-1.7 \times 10^{308} \sim 1.7 \times 10^{308}$	64

例

```
#include <stdio.h>

main()
{
    unsigned char age = 25;
    double height = 166.7;
    float weight = 58.5;

    printf("年齢：%d歳¥n", age);
    printf("身長：%fcm¥n", height);
    printf("体重：%fkg¥n", weight);
}
```

整数型：unsigned char age ／ 整数値：25
実数型：double height、float weight ／ 実数値：166.7、58.5
変数の宣言と代入（初期化）
処理

実行結果

```
年齢：25歳
身長：166.700000cm
体重：58.500000kg
```

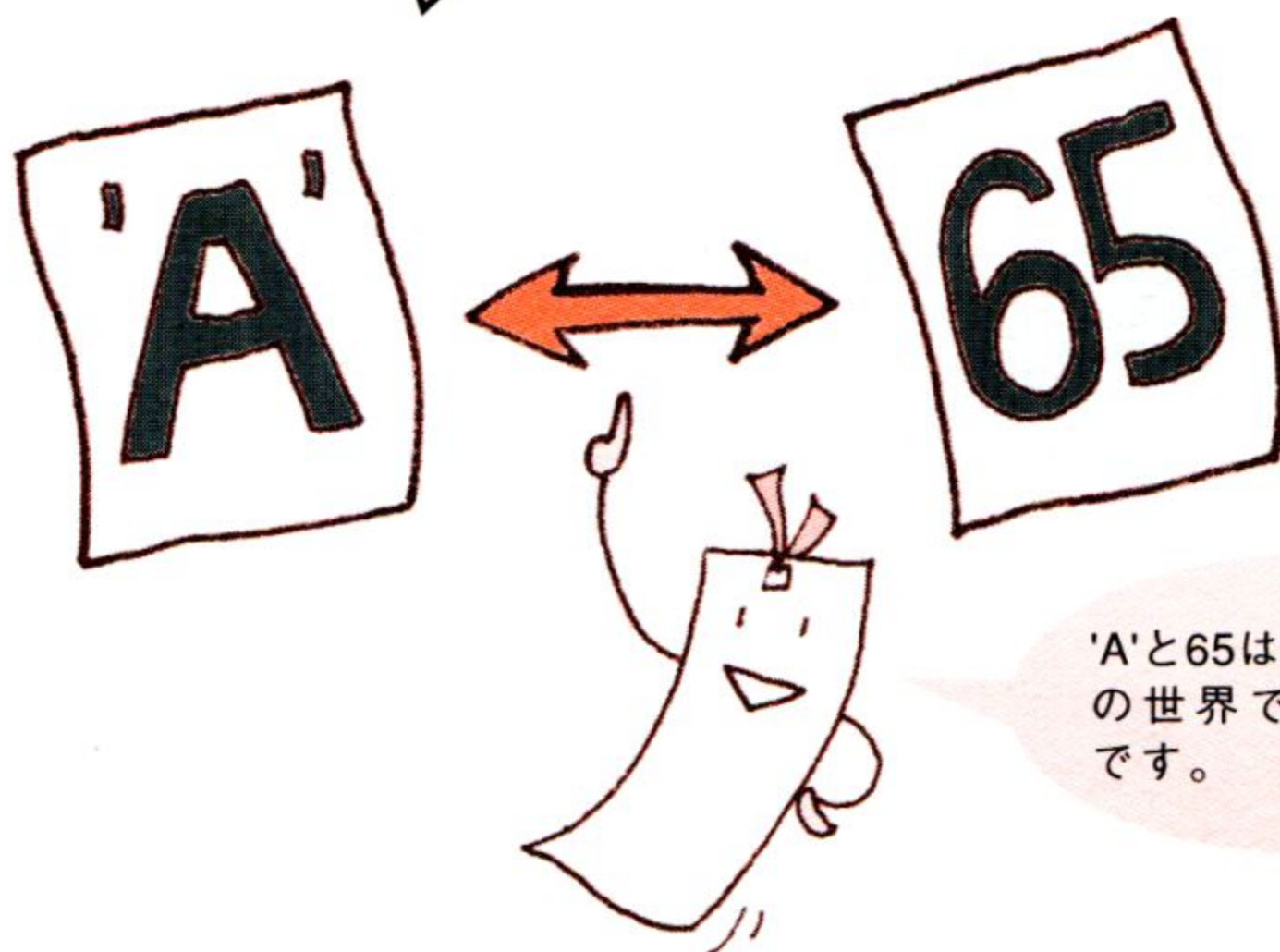
すべての宣言は処理よりも前に書きます。

文字型

ASCIIコードと文字の関係や、文字型変数の使い方について見ていきます。

ASCIIコード

コンピュータでは文字をそのまま文字として扱うことはできません。英数字などの文字をそれぞれ0～127の番号に対応させて管理しています(記述するときは ' でくくる)。その対応を示した国際標準の表のことをASCIIコード表といいます（詳細は巻末付録参照)。

No	文字	No	文字	No	文字	No	文字
32		48	0	64	@	80	P
33	!	49	1	65	A	81	Q
34	"	50	2	66	B	82	R
35	#	51	3	67	C	83	S
36	$	52	4	68	D	84	T
37	%	53	5	69	E	85	U
38	&	54	6	70	F	86	V

ASCIIコード表

文字型

C言語で「文字」とは、半角文字1個のことです。この「文字」を格納するための変数の型が、文字型charです。charは−128〜127の整数が入る型でしたが、C言語では文字と文字コード（0〜127番）を同等と見なしますので、文字を格納する型としても流用できるのです。

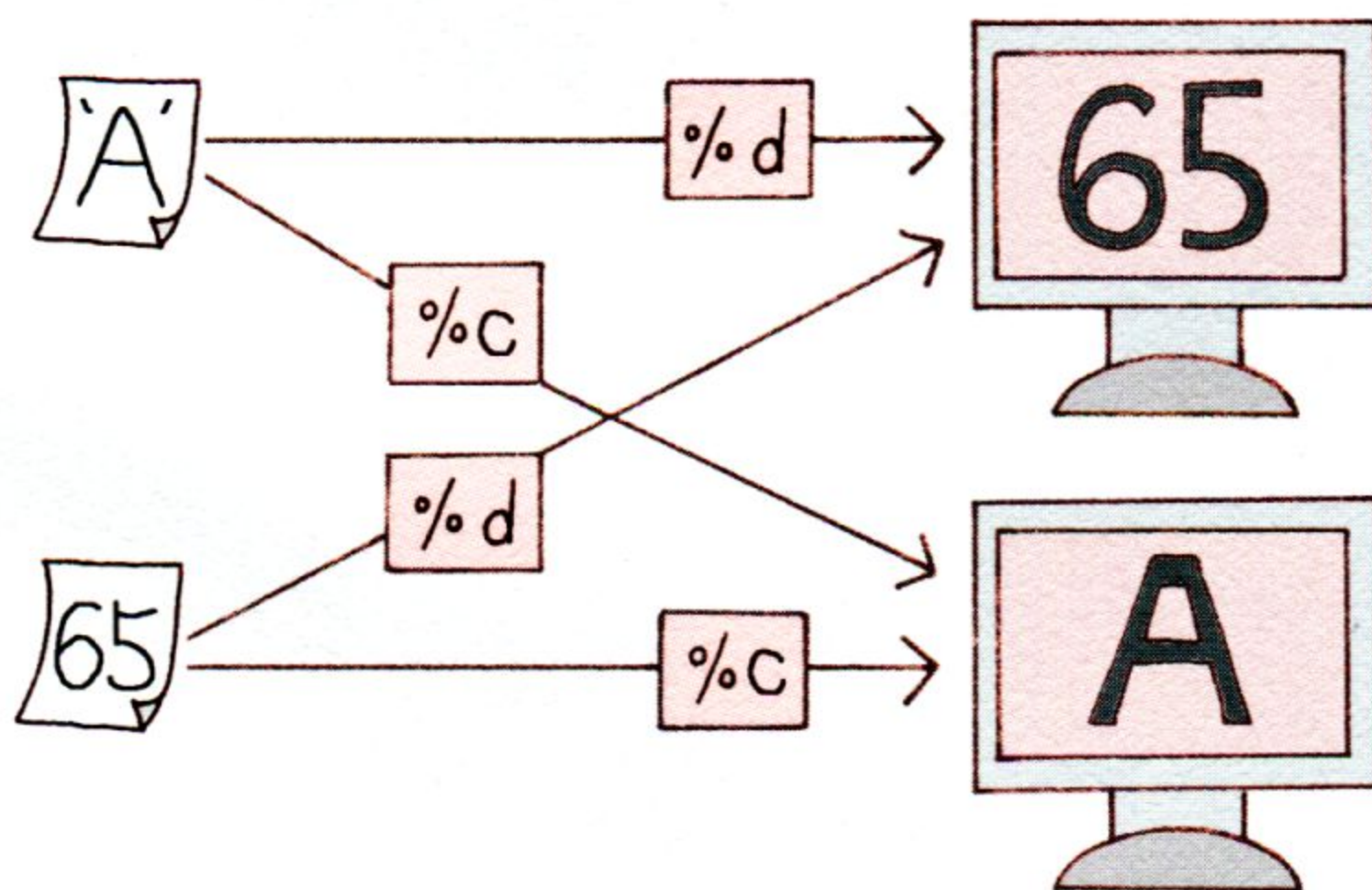
例

```
#include <stdio.h>

main()
{
    char a = 'A';
    printf("%d¥n", a);
    printf("%c¥n", a);
}
```

'A'と65は同等なので、char a = 65;
と書いても同じです。

実行結果

```
65
A
```

65 ← 文字コードとして表示
A ← 文字として表示

複数の文字を1つの文字型変数に代入することはできません。半角文字1個だけです。また、漢字などの全角文字も内部的には複数の文字で表されるため、文字型変数には入りません。

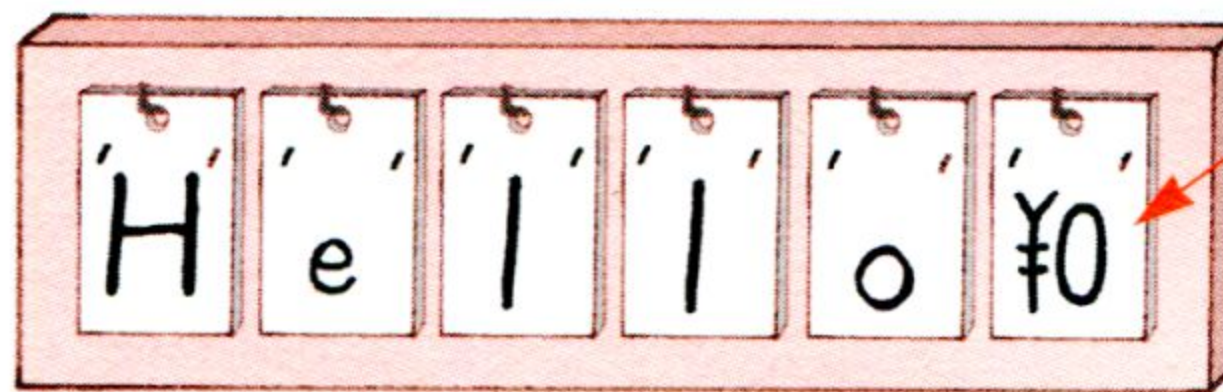
```
char a = "ABC";
char a = "あ";
```

文字列

文字列は文字の集まりです。C言語における文字列のしくみを見ていくことにします。

文字列のしくみ

C言語において、文字列は文字の集まり（**配列**→66ページ）で表されます。記述するときは " でくくります。固定の文字列は次のようなしくみになっています。

"Hello"

'H' 'e' 'l' 'l' 'o' '¥0'

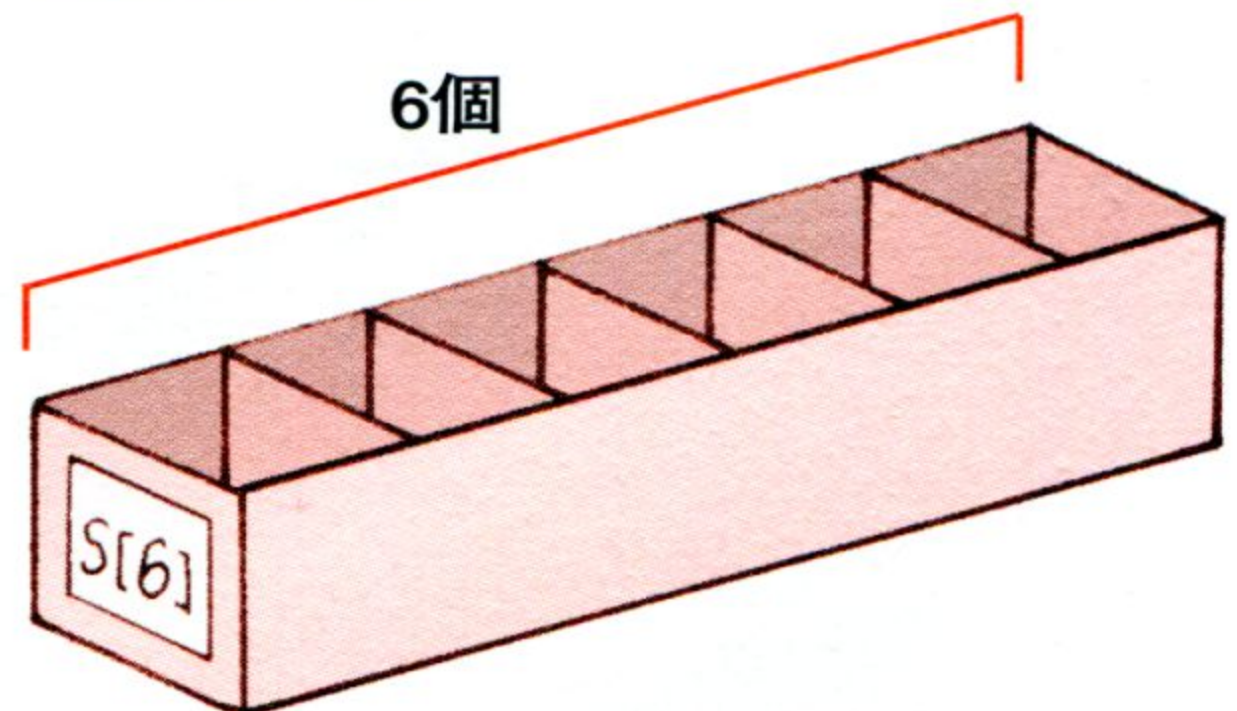
NULL文字
文字列がここで最後であることを表します。画面には表示されません（'¥0'で1文字分）。

文字列を格納する変数を用意するには次のように宣言します。

```
char s[6];
```

変数名　文字列の長さにNULL文字1つ分を加えた数以上を指定。

6個

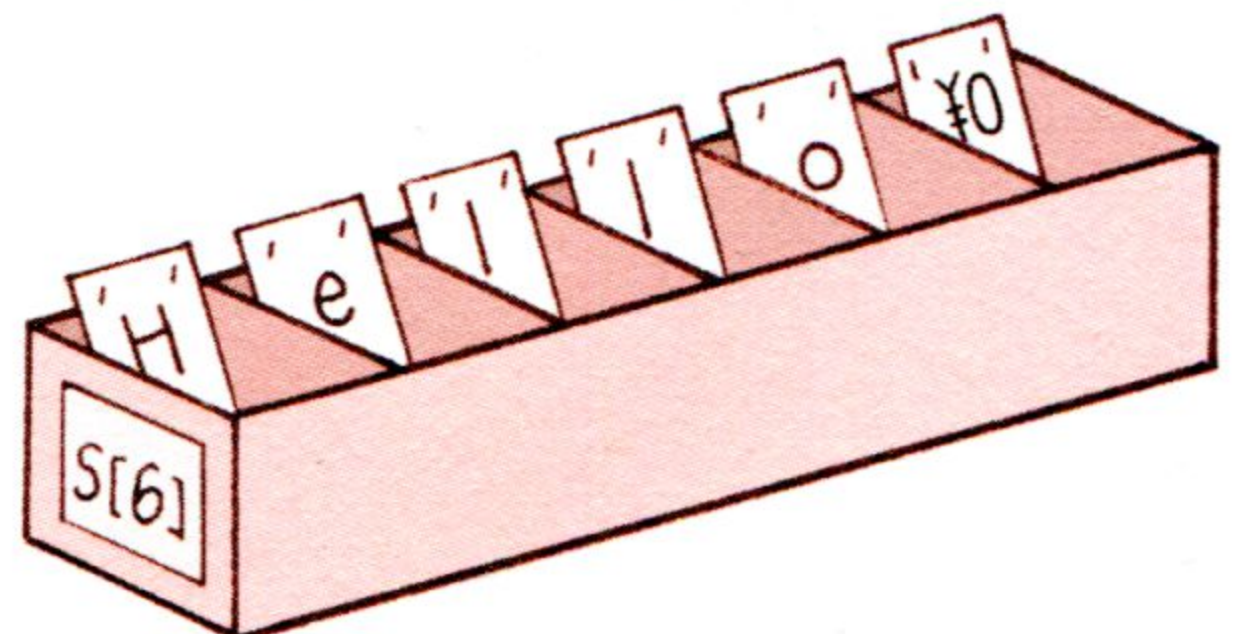
S[6]

文字列を初期化するには次のようにします。

```
char s[6] = "Hello";
```

[ ] の中身を省略すると、文字数＋1個（6個）分の箱を自動的に用意できます。

```
char s[] = "Hello";
```

'H' 'e' 'l' 'l' 'o' '¥0'

S[6]

C 文字列を変数に代入する

文字列の変数に値を入れるとき、「=」が使えるのは初期化時だけです。それ以外のケースで代入するときは、**strcpy**(ストリングコピー)()関数を使います。

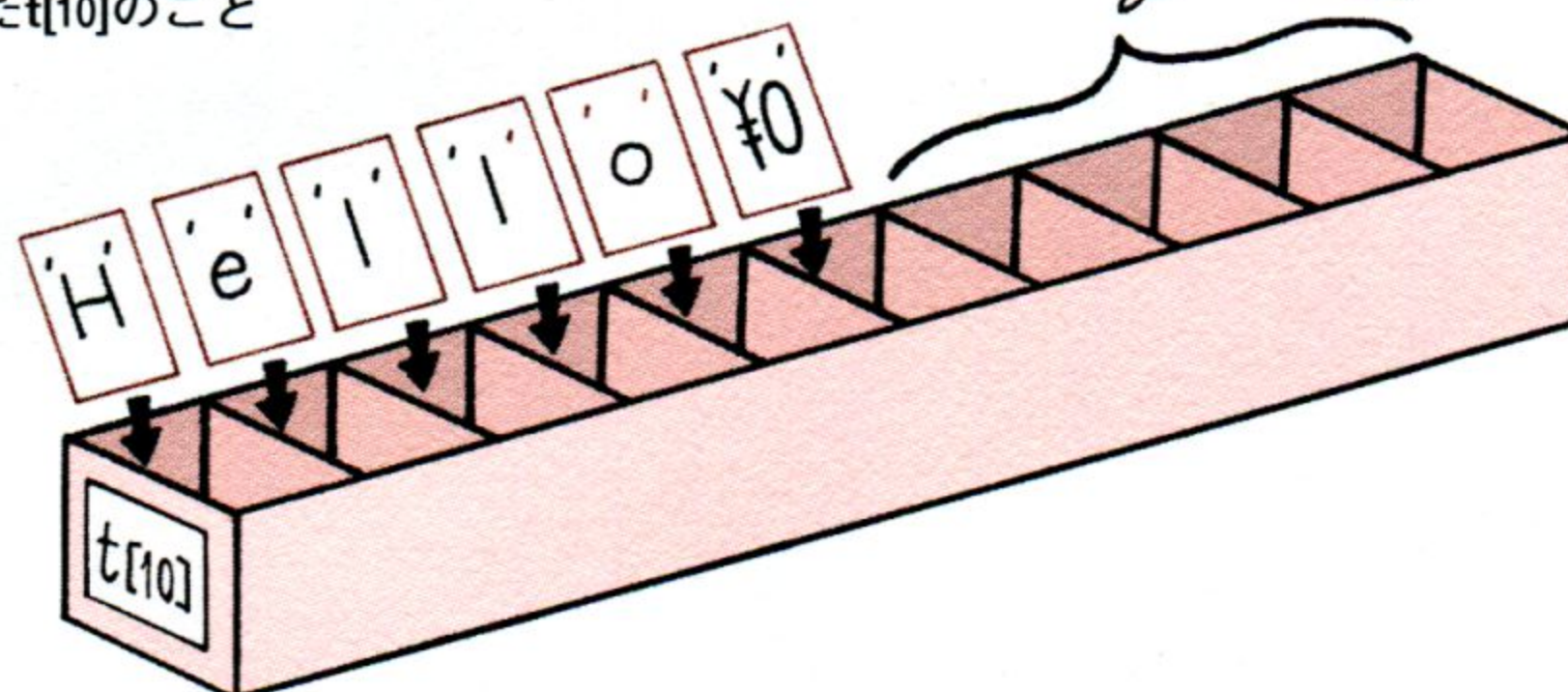
```
char t[10];
strcpy(t, "Hello");
```

上の行で宣言したt[10]のことを指します。

この部分は空きになります。

H e l l o ¥0

t[10]

例

```
#include <stdio.h>
#include <string.h>

main()
{
    char s[10] = "Hello";
    printf("%s¥n", s);
    strcpy(s, "Good bye");
    printf("%s¥n", s);
}
```

strcpy( )を使うために必要です。

あとで "Good bye"を代入するために10文字分用意します。

実行結果

```
Hello
Good bye
```

sの初期値"Hello"

あとで代入した"Good bye"

これまでで、'A'と"A"の違いはわかりましたか？ これらの引用符の使い方は重要ですので、今のうちに理解しておきましょう。

printf( )の書式指定

printf()の書式指定フィールドで指定できる書式と、¥nのように特殊な処理をする制御文字を紹介します。

桁数の指定

printf()の書式指定で%dを指定すると整数を表示できましたが、次のようにして桁数を指定することもできます。

空白を含めて4文字で表示

```
printf("%4d", 25);
```

```
  25
```

4桁

0を使って4文字で表示

```
printf("%04d", 25);
```

```
0025
```

4桁

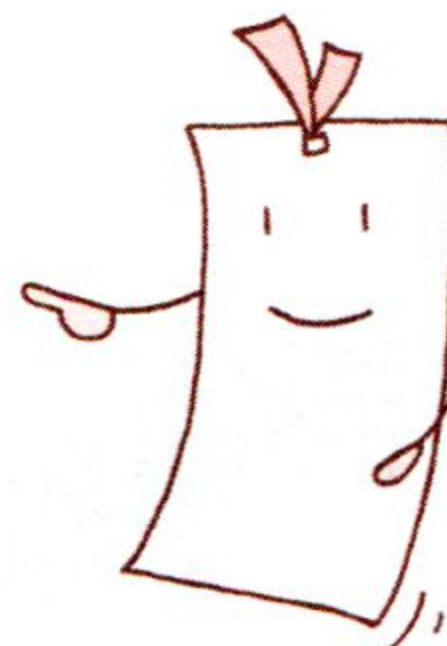
実数を表示する%fでは小数点前後の桁数を指定できます。

全体を6桁、小数点以下を1桁で表示

```
printf("%6.1f", 155.32);
```

```
 155.32
```

表示されません。（2）

6桁

小数点も1文字と数えます。

文字列についても同様に、表示位置をそろえることができます。

全体を6文字として表示

```
char name[] = "Akira";
printf("%6s", name);
```

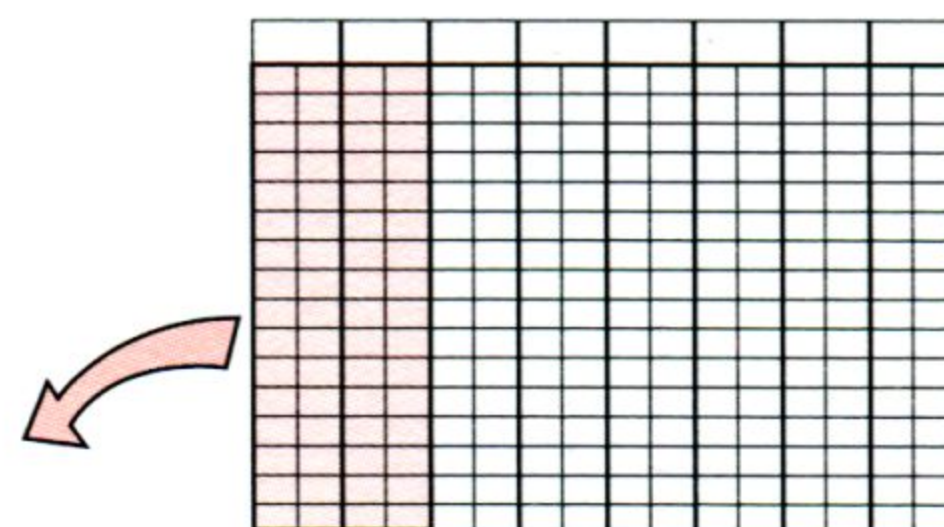
```
 Akira
```

6文字

C 特殊な動作を表す文字

¥nのような、¥（エスケープ文字）ではじまる2文字のことを**エスケープシーケンス**といいます。これらの文字は画面上に表示されず、次のように特殊な動作を表します。

コード	エスケープシーケンス	働き
0	¥0	ヌル文字(NULL)
8	¥b	バックスペース(BS)
9	¥t	タブ(TAB)
10	¥n	改行(LF)
13	¥r	復帰(CR)

ASCIIコード表

ASCIIコードの若い番号に割り当てられています。

¥自身を表示させたいときは¥¥と書きます。文字列や文字の引用符である ' や " を表示するにはその前に¥をつけます。

書き方	表示
¥¥	¥
¥'	'
¥"	"

例

```
#include <stdio.h>

main()
{
    printf("     %8s %8s¥n", "商品A", "商品B");
    printf("数量 %08d %08d¥n", 16, 246);
    printf("重量 %8.4f %8.4f¥n", 76.3, 556.1);
    printf("%d%c", 20, 10);
    printf("%d¥bA¥n", 20);
    printf("%d¥t%d¥n", 20, 30);
}
```

実行結果

```
         商品A       商品B
数量    00000016    00000246
重量    76.3000     556.1000
20
2A
20      30
```

表形式にまとめています。

文字コード10は改行になります。

最後の0が消えてAになります。

20と30の間にタブがあります。

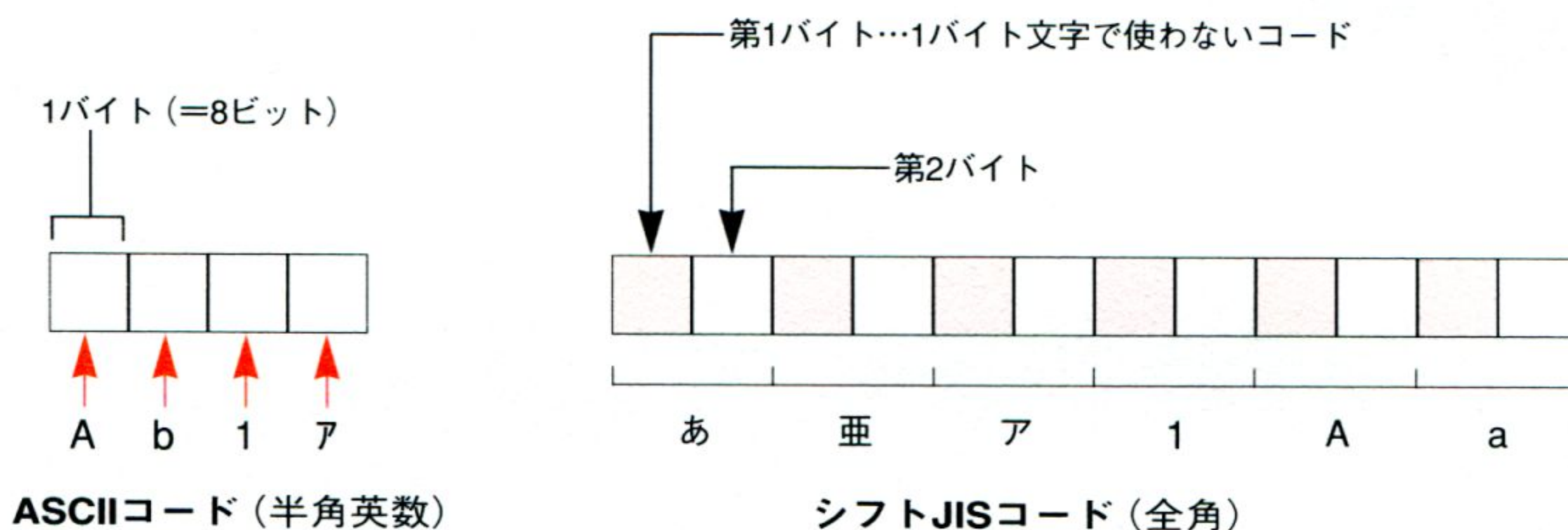
COLUMN コラム

～日本語について～

ASCIIコードでは0～127の数字を使います。欧米ではアルファベットと数字、記号を合わせても100種類程度で済むため、7ビット（→36ページ）で表現するという決まりができたのです。たしかに、英数字だけであれば１バイト（=8ビット）の記憶容量があれば十分です。しかし、日本語は漢字の数が多いため、これではとても足りません。

そこで、日本語は1バイト以上使って表します。現在、多くのパソコン（Windows/Macintosh系）では文字を表すときに「シフトJIS」というコードを使っています。このコードでは半角文字を表すのに1バイト、全角文字を表すのに2バイトを必要とします。

1バイト（=8ビット）
A　b　1　ｱ
ASCIIコード（半角英数）

第1バイト…1バイト文字で使わないコード
第2バイト
あ　亜　ア　１　Ａ　ａ
シフトJISコード（全角）

1バイトでは256種類の情報しか表すことができませんが、2バイト使えば、第1バイトと第2バイトの組み合わせで65,536（256×256）種類の文字コードを表せます。第1バイトに文字としては使わないコードを指定することで、「このバイトと次のバイトは合わせて1つの文字（2バイト文字）を表しています」と表現しているのです。

また、インターネットのメールなどで使われている日本語コードは「JISコード」といいます。JISコードは1バイトのうち7ビットを使い、はじまりにシフトイン、おわりにシフトアウトという特殊なコードを用いて他のコードと区別しています。

他にも、日本語を表すコード体系にはUNIXで使われるEUC（イーユーシー）コードや、日本語・アルファベットを問わず、すべての文字を2バイトで表すUNICODE（ユニコード）などがあります。

`strcpy()`など文字列を処理する関数は、たいてい英語用にできています。日本語コードの細かい仕様はともかく、単純に英語と同じように扱えないということだけは覚えておいてください。

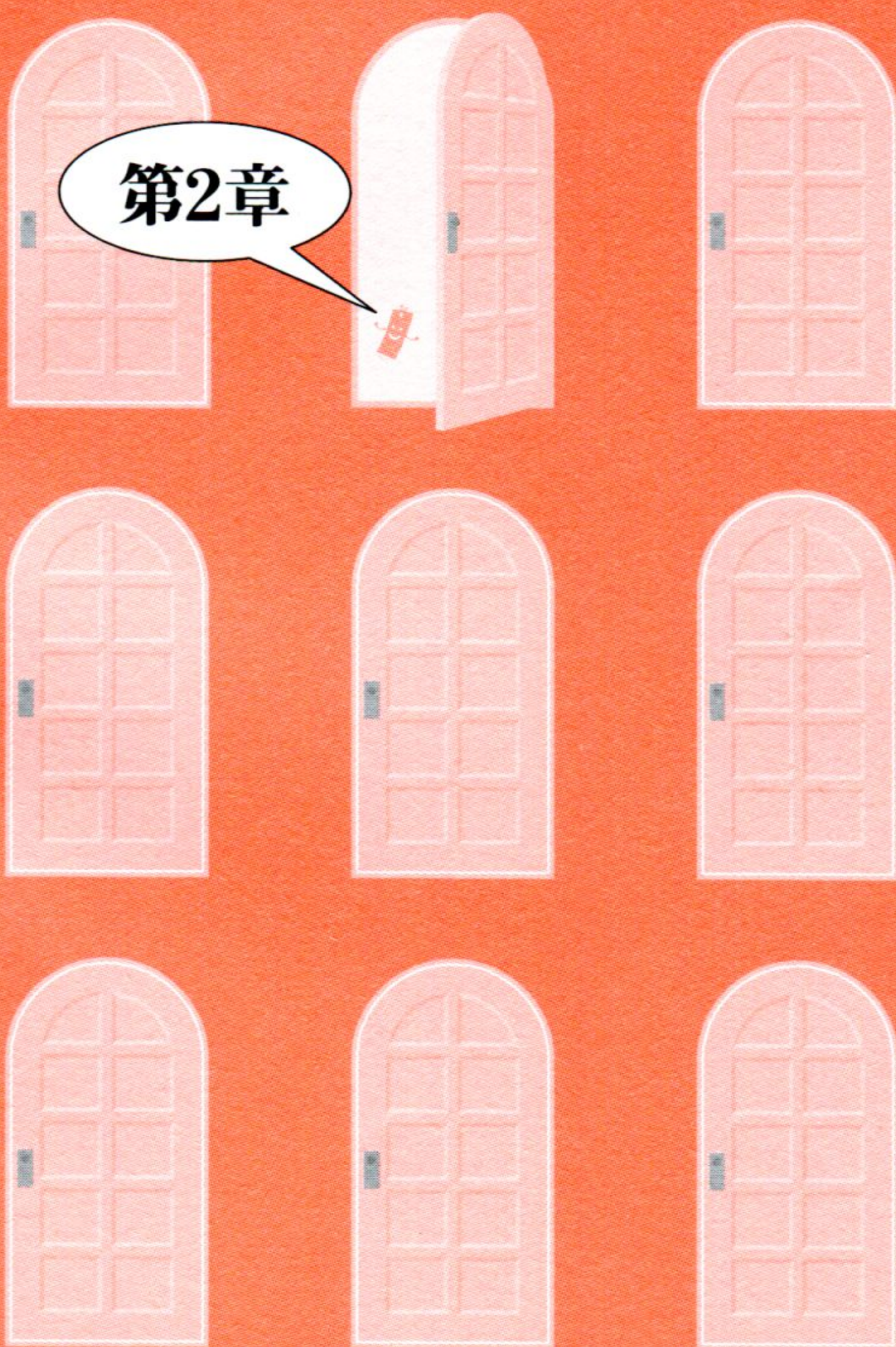
2

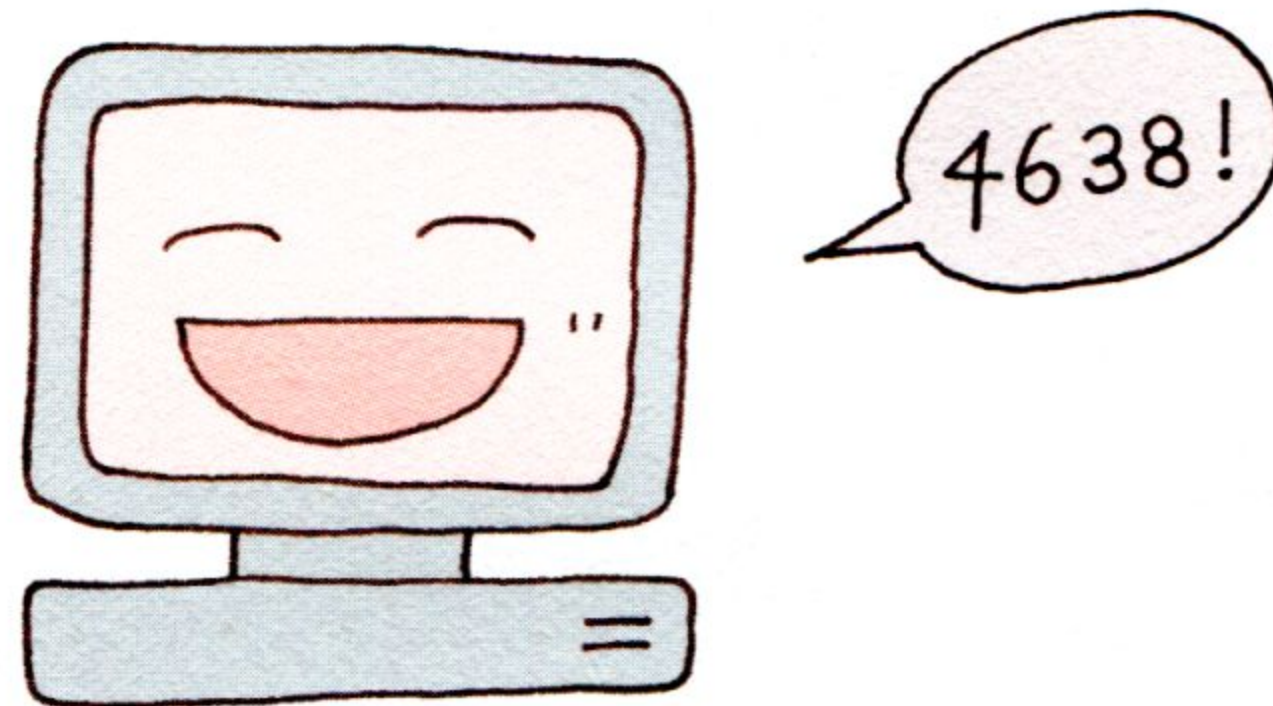
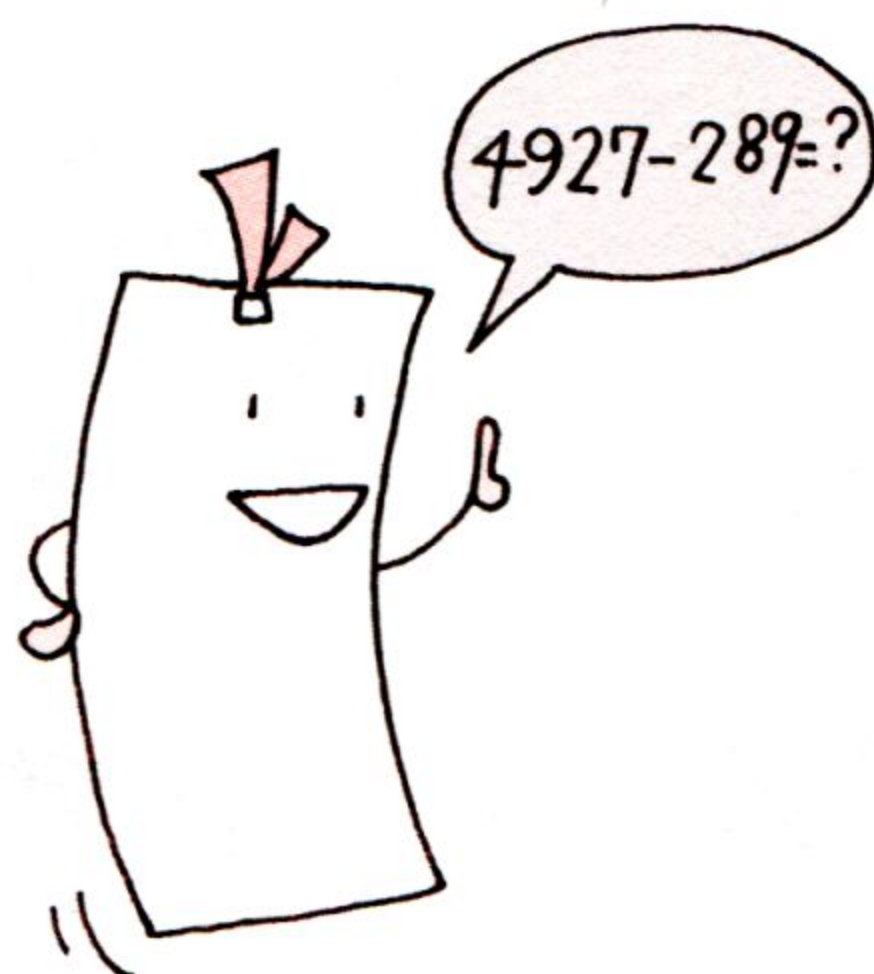
演算子

第2章

第2章は

ここがkey!

Topics

コンピュータが計算機代わりに

この第2章では**演算子**について学びます。演算子とは要するに、計算で使う「＋」や「−」記号のことです。ただし、コンピュータのキーボードに「÷」がないことでもわかるように、数学で使う演算子とはちょっと書き方が違うものがあります。また、コンピュータの計算は算術演算だけではありません。

まず、紹介するのが数値計算を行うときに使う演算子です。ここでは算数の教科書で見たことのある、おなじみの記号が登場します。たとえば、コンピュータに足し算をしてもらいたいときに使う「+（プラス）」や引き算をしてほしいときに使う「-（マイナス)」、これらも立派な演算子です。他にもかけ算や割り算、変わったところでは割り算の余りを出す演算子なんていうのもあります。

演算子は、計算を行うものばかりではありません。C言語にはコンピュータならではの様々な働きをする演算子がたくさんあります。値を比べるときに使う**比較演算子**、条件判断のときに使う**論理演算子**などがそれです。

いろいろな値を入れてみて計算結果を試せるので、第1章のプログラムよりはコンピュータとの対話を楽しめるのではないかと思います。

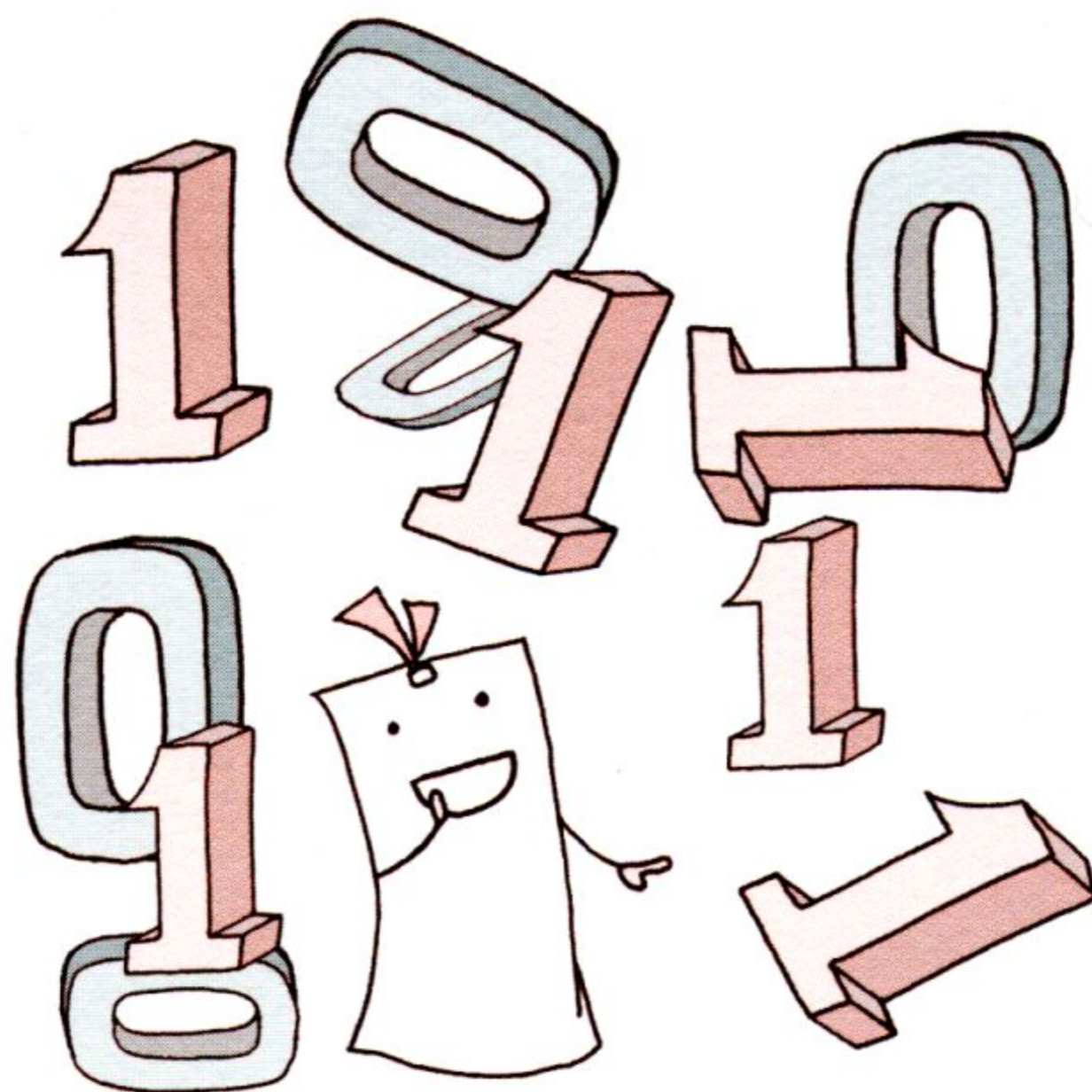
Topics C

コンピュータは1か0かのデジタルな世界

また、巷でよく聞く**ビット**や**バイト**って何だろう、という疑問にも答えます。

ビットやバイトを正確に理解するには、**2進数**や**16進数**について知っておく必要があります。私たちは普段10ごとに位の上がる10進法を使っていますね。これは、とりもなおさず人間の指が10本だからです。一方、コンピュータ内では1か0、つまりオンかオフの2通りのデータしか存在しません。すべての情報は1と0の組み合わせ（2進数）で表されます。

しかし、昔の穴空きテープを読める博士ならともかく、私たちが1と0だけ見て、どんなデータかを把握するのはかなり面倒です。それでなくとも1と0しか使えないのでは冗長になりすぎます。そこで、今では16進数を使うのが一般的になっています。これは2進数を4桁ずつで区切り、この4桁で表せる2^4=16通りの数を0～9とA～F（10進数の10～15に相当）の文字を使って表す方法です。これにより、2進数のデータが私たちにとって少しだけわかりやすいものになります。

演算子はプログラムの要です。先に進むにつれて難易度もあがってきますが、あせらず、ひとつひとつをきちんと理解してから次に進んでいきましょう。

計算の演算子(1)

計算に用いる＋や－などのことを演算子といいます。演算子を使って実際に計算してみましょう。

数の計算で使う演算子

C言語で数の計算に用いる演算子には次のものがあります。

演算子	働き	使い方	意味
プラス +	＋（足す）	a = b + c	bとcを足した値をaに代入
マイナス -	―（引く）	a = b - c	bからcを引いた値をaに代入
アスタリスク *	×（かける）	a = b * c	bとcかけた値をaに代入
スラッシュ /	÷（割る）	a = b / c	bをcで割った値をaに代入 （cが0のときはエラー）
パーセント %	…（余り）	a = b % c	bをcで割った余りをaに代入 （整数型でのみ有効）
イコール =	＝（代入）	a = b	bの値をaに代入

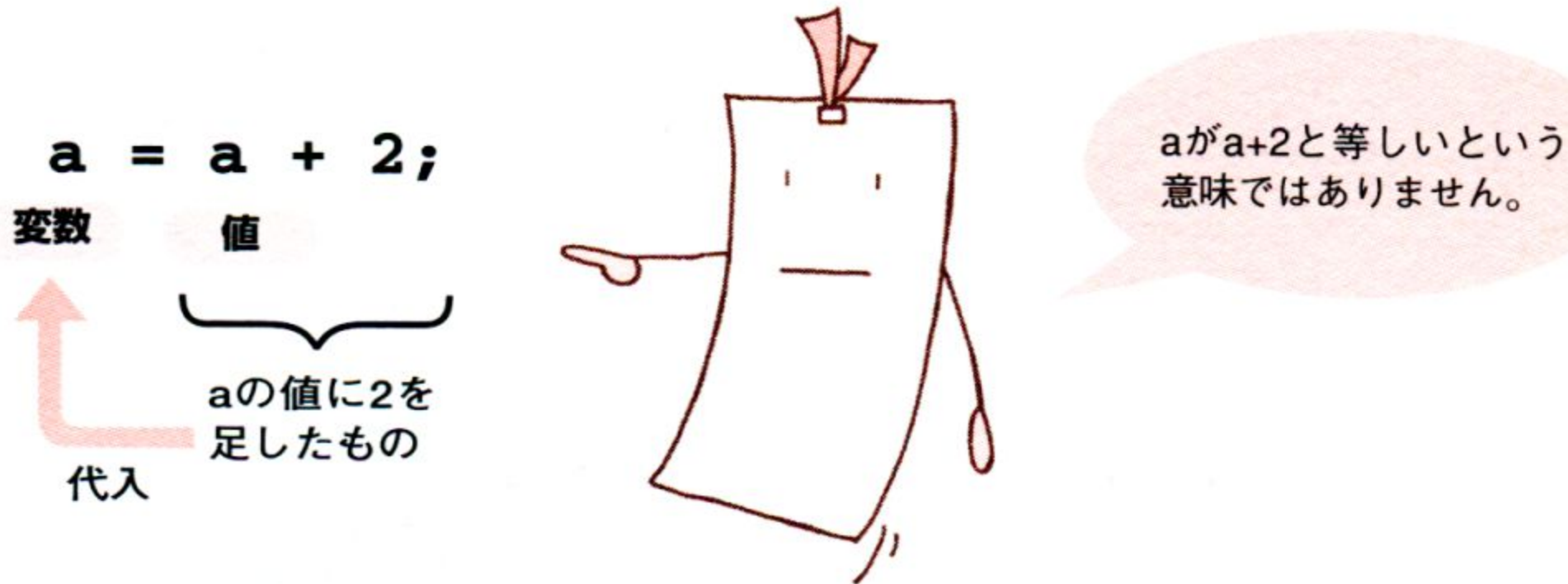
例

```
#include <stdio.h>

main()
{
    printf("5＋5は%dです。¥n", 5+5);
    printf("5―5は%dです。¥n", 5-5);
    printf("5×5は%dです。¥n", 5*5);
    printf("5÷5は%dです。¥n", 5/5);
    printf("5÷3の余りは%dです。¥n", 5%3);
}
```

実行結果

```
5＋5は10です。
5―5は0です。
5×5は25です。
5÷5は1です。
5÷3の余りは2です。
▌
```

代入演算子

変数に値を代入する演算子「=」では左辺を変数、右辺を値と見なします。よって、int型の変数aそのものの値を2増やしたいときは次のように書きます。

aの値を2増やすことを、次のように書くこともできます。

```
a += 2;
```

「=」や「+=」を**代入演算子**といいます。代入演算子には他に次のようなものがあります。

演算子	働き	使い方	意味
+=	足して代入	a += b	a+bの結果をaに代入（a = a+bと同等）
-=	引いて代入	a -= b	a-bの結果をaに代入（a = a-bと同等）
*=	かけて代入	a *= b	a*bの結果をaに代入（a = a*bと同等）
/=	割って代入	a /= b	a/bの結果をaに代入（a = a/bと同等）
%=	余りを代入	a %= b	a%bの結果をaに代入（a = a%bと同等）

例

```
#include <stdio.h>

main()
{
    int a = 90;

    a += 10;
    printf("90に10を足したら%dです。¥n", a);
}
```

a = a+10;と書いても同じです。

実行結果

```
90に10を足したら100です。
```

計算の演算子(2)

値を1増やすインクリメント演算子と1減らすデクリメント演算子について解説します。

インクリメント・デクリメント演算子

インクリメント（加算）演算子、デクリメント（減算）演算子は、整数型の変数の値を1つ増やしたり減らしたりする場合に使います。

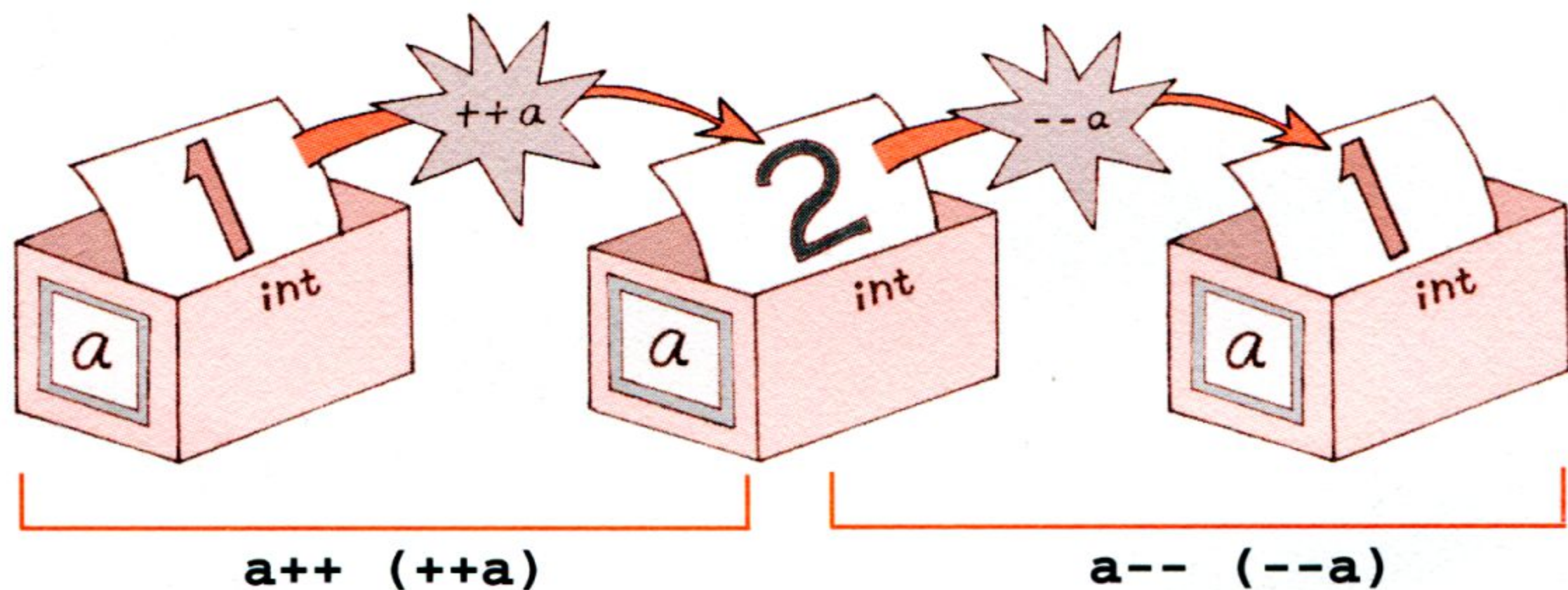
演算子	名称	働き	使い方	意味
++	インクリメント演算子	変数の値を1増やす	a++または++a	aの値を1増やす
--	デクリメント演算子	変数の値を1減らす	a--または--a	aの値を1減らす

++a

--a

a++ （++a）

a-- （--a）

例

```
#include <stdio.h>

main()
{
    int a = 1;
    printf("はじめは%dでした。¥n", a);

    a++;
    printf("1増えて%dになりました。¥n", a);

    a--;
    printf("1減って%dに戻りました。¥n", a);
}
```

実行結果

```
はじめは1でした。
1増えて2になりました。
1減って1に戻りました。
```

≫a++と++aの違い

インクリメント・デクリメント演算子には、それぞれ2種類の書き方があり、++a(--a)の方を前置（ぜんち）、a++(a--)の方を後置（こうち）といいます。
前置と後置では演算を行うタイミングが異なり、前置の場合は変数の参照より先に、後置の場合は変数の参照より後に演算が行われます。そのため、次のようなことが起こります。

```
int x, a = 1;
x = ++a;
```

aに1を足した後、xに値を代入する
→ xの値は2になる

```
int x, a = 1;
x = a++;
```

xに値を代入した後、aに1を足す
→ xの値は1のまま

例

```
#include <stdio.h>

main()
{
    int a = 1, b = 1;

    printf("前置だと%dになります。¥n", ++a);
    printf("後置だと%dになります。¥n", b++);
}
```

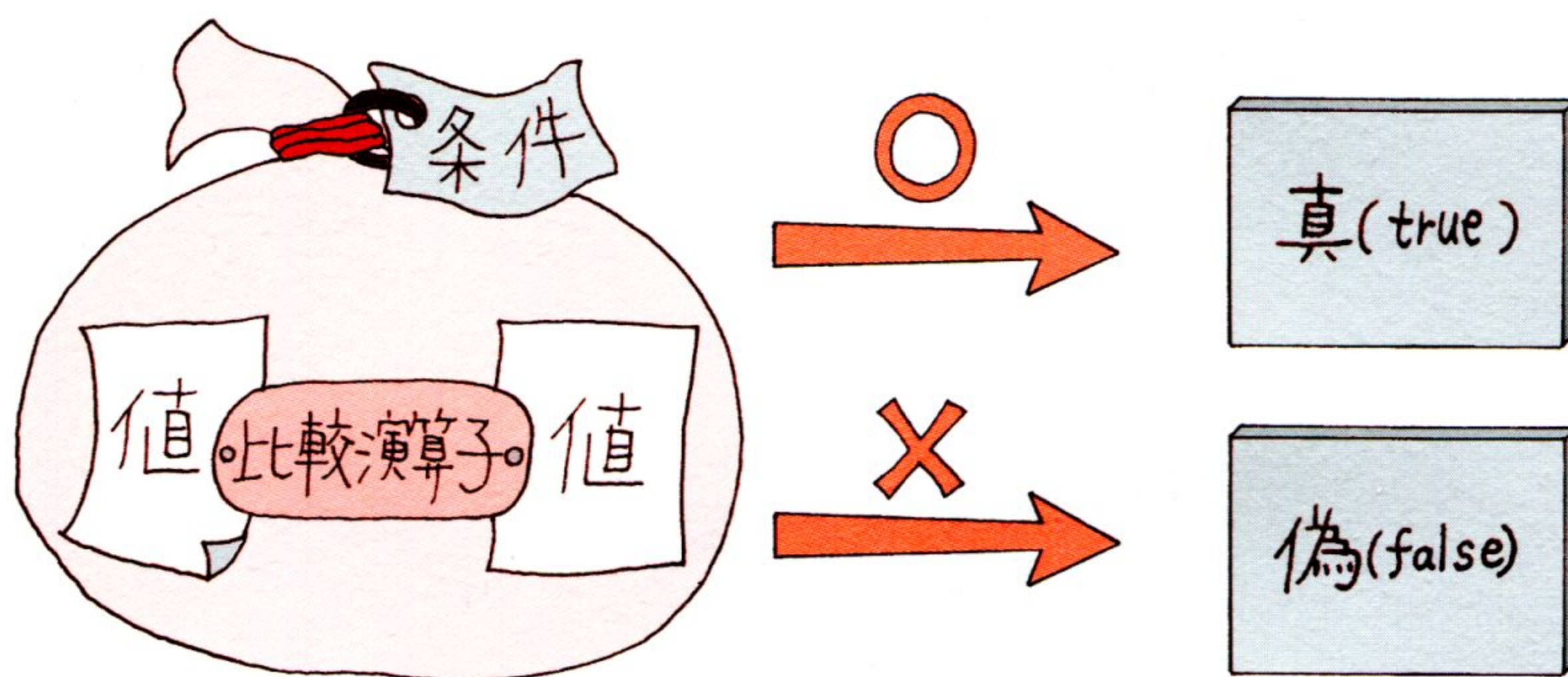
実行結果

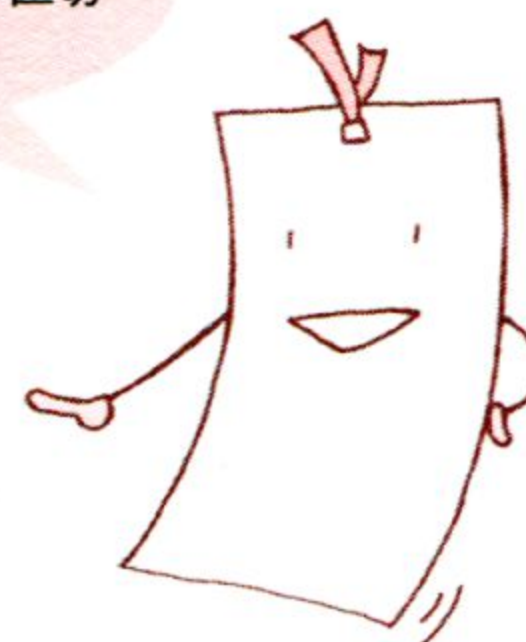
```
前置だと2になります。
後置だと1になります。
```

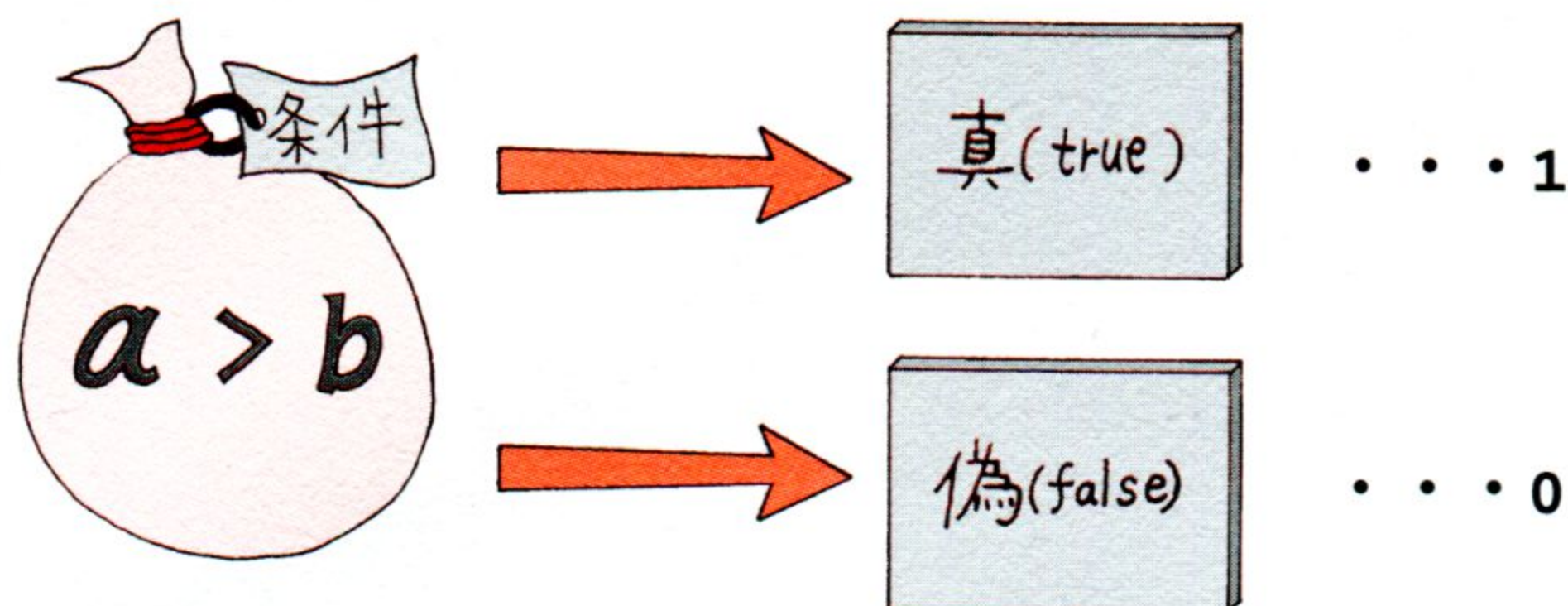
比較演算子

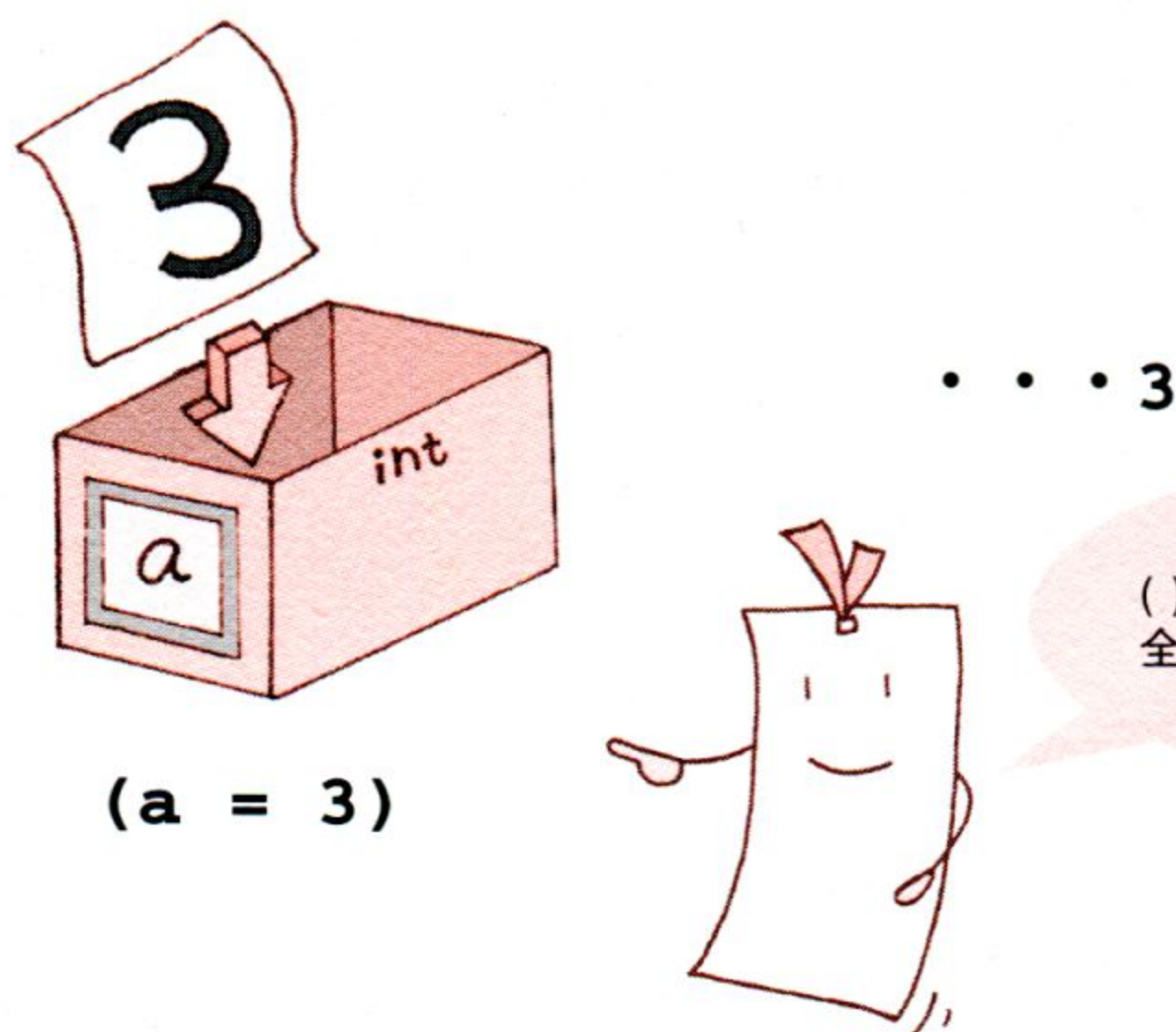
条件式を作るときに使う比較演算子を紹介します。

比較演算子とは？

C言語では変数や数などの値を比較して、条件式を作り、その結果によって処理を変えることができます。このとき使う演算子を比較演算子といいます。
条件が成立した場合を「**真**（true）」、成立しない場合を「**偽**（false）」といいます。

2つの記号で1つの働きをしているものはスペースなどで区切らないでください。

演算子	働き	使い方	意味
==	＝（等しい）	a == b	aとbは等しい
<	＜（小なり）	a < b	aはbより小さい
>	＞（大なり）	a > b	aはbより大きい
<=	≦（以下）	a <= b	aはb以下
>=	≧（以上）	a >= b	aはb以上
!=	≠（等しくない）	a != b	aとbは等しくない

C 式が持っている値

条件式や代入式はそれ自体が値を持っています。
たとえば、条件式が偽であるとき、条件式そのものは0という値を持ちます。条件式が真のときは条件式は1になります。

代入式では、代入した値がその代入式全体の値になります。

例

```
#include <stdio.h>

main()
{
    int a = 10, b = 20;

    printf("a=%d b=%d¥n", a, b);
    printf("a<b ・・・ %d¥n", a<b);
    printf("a>b ・・・ %d¥n", a>b);
    printf("a==b ・・・ %d¥n", a==b);
    printf("a=b ・・・ %d¥n", (a=b));
}
```

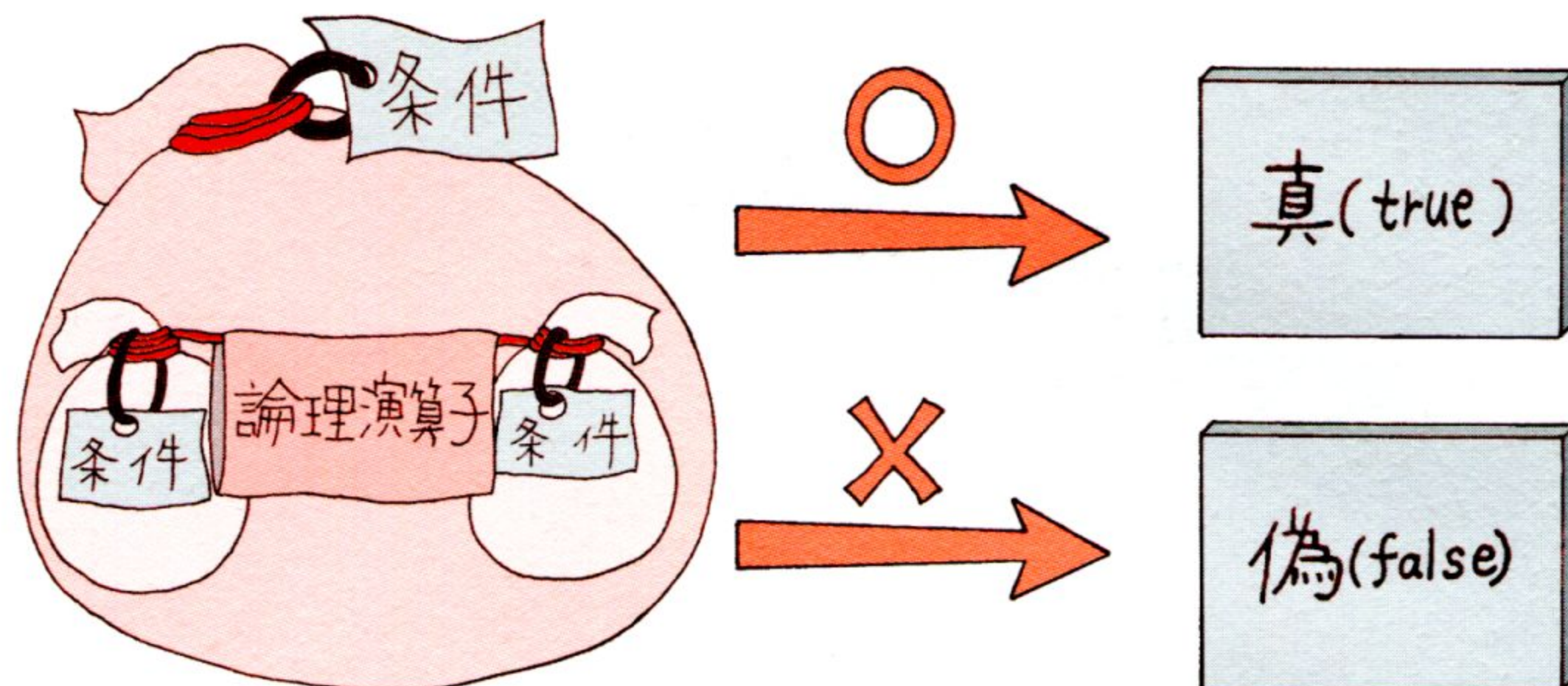
実行結果

```
a=10 b=20
a<b ・・・ 1
a>b ・・・ 0
a==b ・・・ 0
a=b ・・・ 20
```

論理演算子

複数の条件式を組み合わせて、より複雑な条件式を作ることができます。

論理演算子とは？

複数の条件を組み合わせて、より複雑な条件を表すときに使うのが論理演算子です。

論理演算子には次の3種類があります。

<table>
<tr><th>演算子</th><th>働き</th><th>使い方</th><th>意味</th></tr>
<tr><td>&&</td><td>かつ</td><td>(a >= 10) && (a < 50)</td><td>aは10以上かつ50未満</td></tr>
<tr><td>||</td><td>または</td><td>(a == 1) || (a == 100)</td><td>aの値が1か100</td></tr>
<tr><td>!</td><td>～ではない</td><td>!(a == 100)</td><td>aは100ではない</td></tr>
</table>

条件A、Bがあるとき、論理演算子の働きを図示すると、次のようになります。

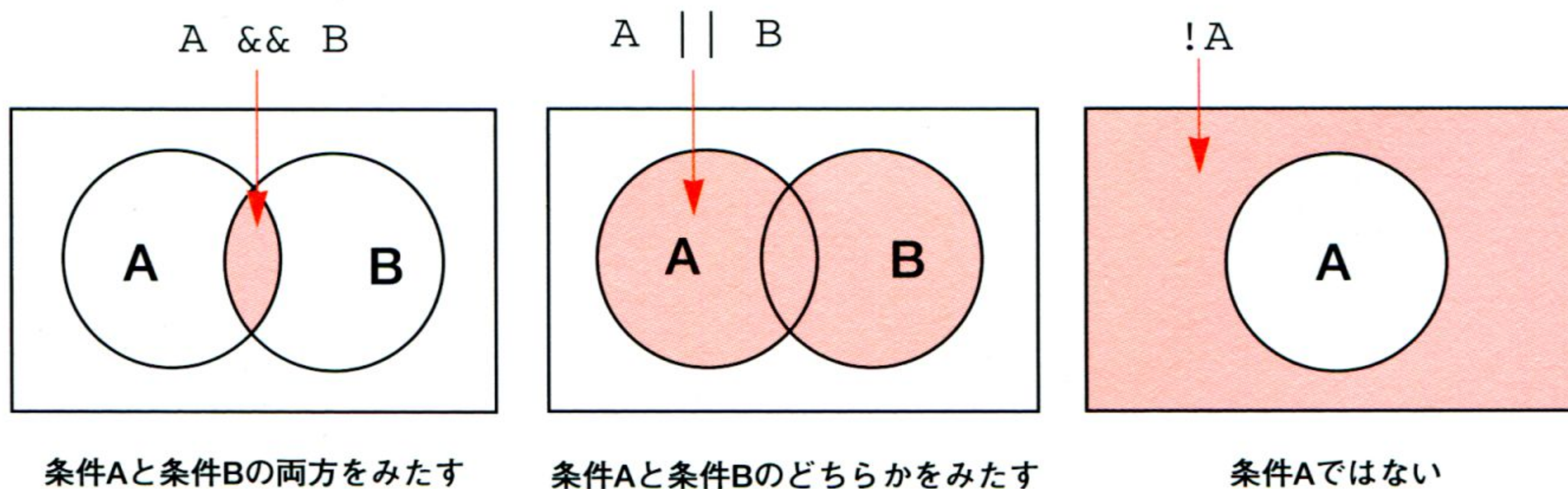
A && B

条件Aと条件Bの両方をみたす

A || B

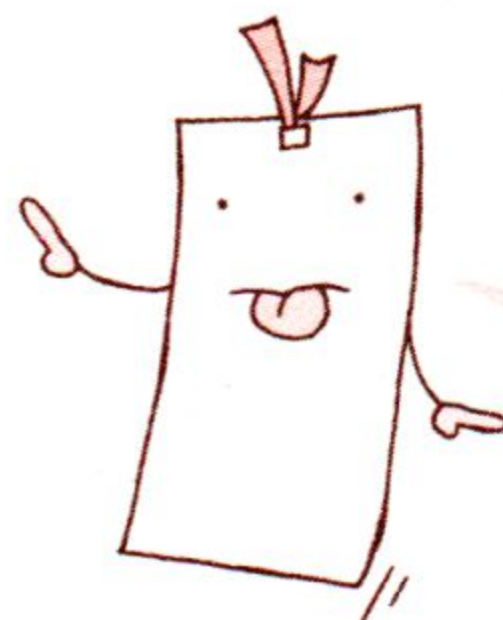
条件Aと条件Bのどちらかをみたす

!A

条件Aではない

≫複雑な条件式

少し複雑な論理演算の例を見てみましょう。各演算子は優先度（→40ページ）に従って処理されますが、意図的に関係をはっきりさせたいときは、()を使います。

aが50以上100未満である

```
(50 <= a) && (a < 100)
```

50<=a<100
とは書けません。

bが0でも1でもない

```
!((b == 0) || (b == 1)) ・・・「b = 0またはb = 1」ではない
!(b == 0) && !(b == 1)  ・・・b = 0ではなく、b = 1でもない
 (b != 0) && (b != 1)   ・・・b ≠ 0かつ、b ≠ 1である
```

C 条件つき代入

?と:の2つの記号を使って、条件によりxに代入する値を変えることができます。
次のように書きます。

```
x = (条件) ? a : b
```

条件が真のときの値
条件が偽のときの値

条件

真(true) … x = a

偽(false) … x = b

0≦a≦100であればaの値を、そうでないときは0を表示します。

例

```
#include <stdio.h>

main()
{
    int a = 30, x;

    x = (0 <= a && a <= 100) ? a : 0;
    printf("%d¥n", x);
}
```

実行結果

```
30
```

n進数

コンピュータの中はオンとオフの2進数の世界です。2進数や16進数について紹介します。

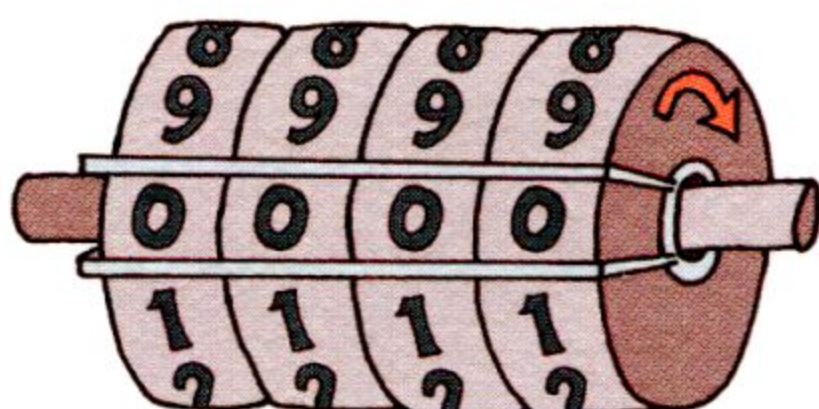
数値の表し方

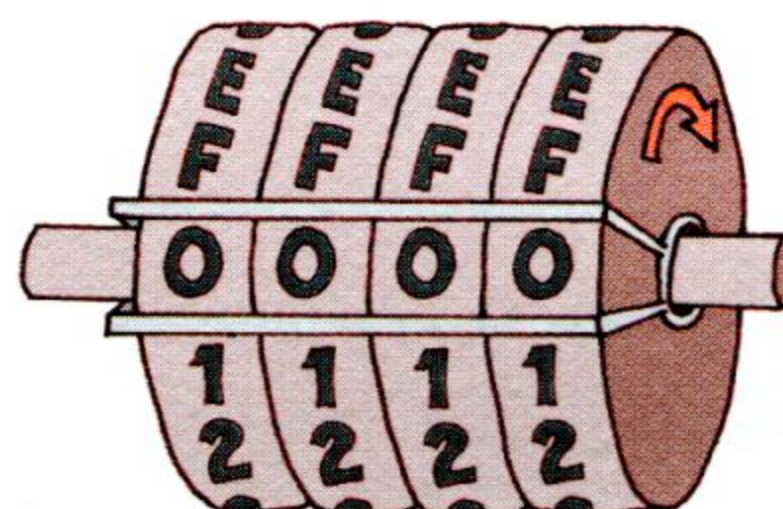
私たちが普段使っている数の表し方は10で位が繰り上がる10進数表記です。しかし、コンピュータの世界では2進数や16進数での表記が一般的です。

2進数

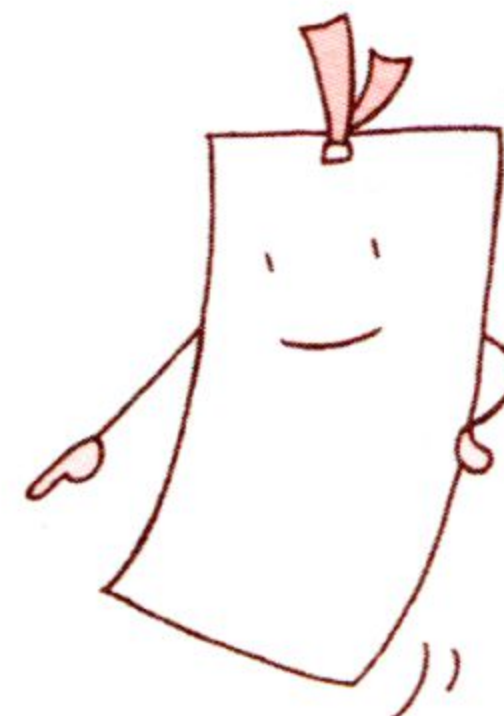
1と0の2つの状態で表します。コンピュータ内部では一番基本的な表し方です。

10進数

普段使っている表記方法で0から9までの数字を使います。

16進数

16で位が繰り上がる表し方です。9のあとはA～Fの文字で表します。

2進数、10進数、16進数表記の関係は次のようになります（ は桁上がり）。

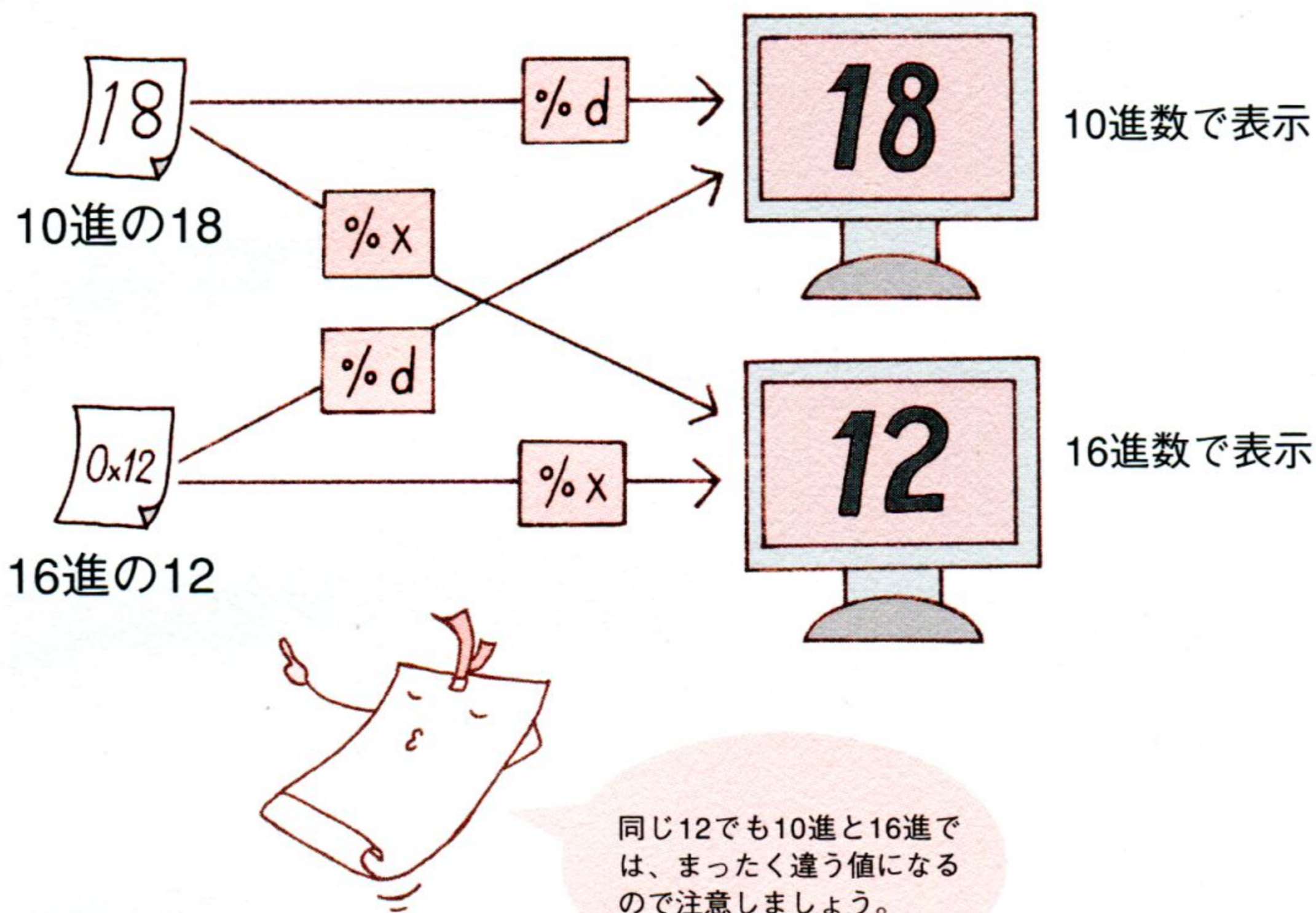
2進数	10進数	16進数
0	0	0
1	1	1
10	2	2
11	3	3
100	4	4
101	5	5
110	6	6
111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F
10000	16	10

2進数と16進数では、10は「ジュウ」ではなく「イチゼロ」と読みます。

C 16進数の表記方法

C言語のプログラムの中で数値を16進数で表記するには、数字の前に0xをつけます。
また、16進で値を表示したい場合にはprintf()の書式指定で%xを使います。

18
10進の18
%d
%x
18
10進数で表示
0x12
16進の12
%d
%x
12
16進数で表示

同じ12でも10進と16進では、まったく違う値になるので注意しましょう。

例

```
#include <stdio.h>

main()
{
    int a = 15, b = 0x11;

    printf("10進数の%dは16進数では%X¥n", a, a);
    printf("16進数の%Xは10進数では%d¥n", b, b);
}
```

大文字で%Xと書くと、16進のA～Fの文字が大文字になります。

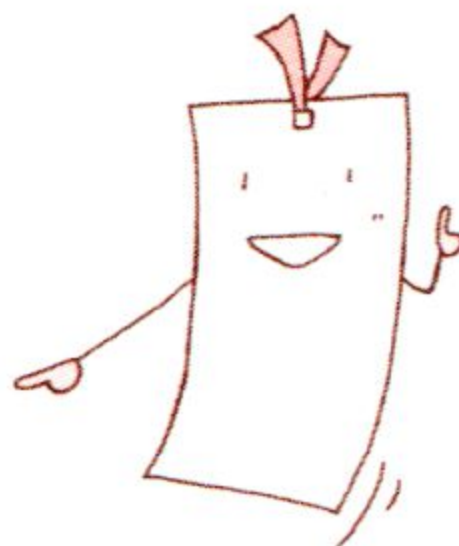
実行結果

```
10進数の15は16進数ではF
16進数の11は10進数では17
```

ビットとバイト

コンピュータで扱うデータの単位である、ビットとバイトを紹介します。

ビットとは？

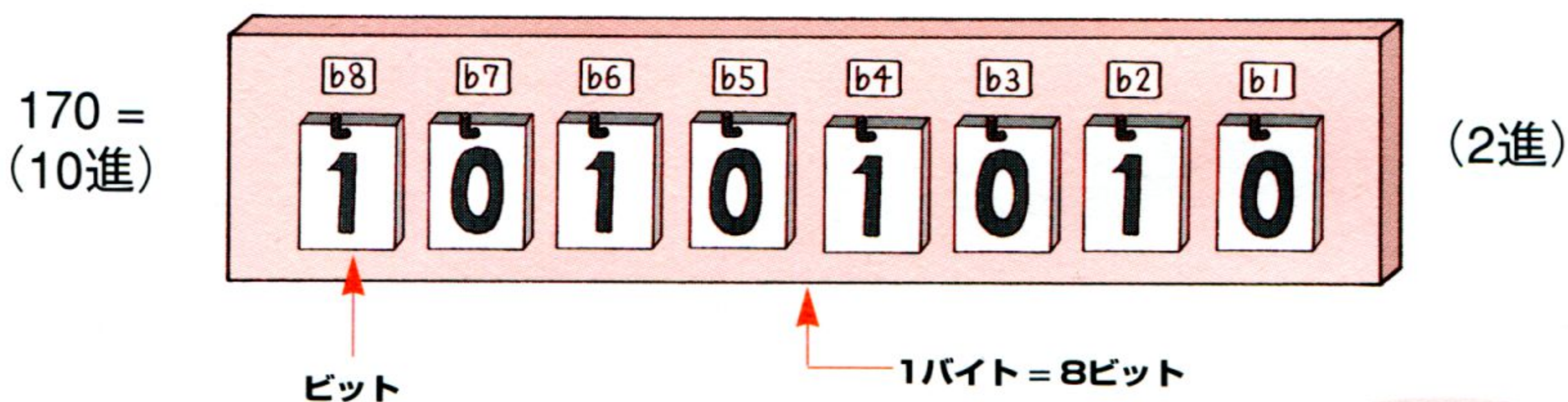
コンピュータで扱う情報は、電気的にオンの状態（1）とオフの状態（0）で表せます。この1か0の値を取る情報の最小基本単位を**ビット**といいます。また、ビットが8つ集まったもの（8ビット）を1**バイト**といいます。

b1を最下位ビット
b8を最上位ビット
といいます。

170 =（10進）

b8	b7	b6	b5	b4	b3	b2	b1
1	0	1	0	1	0	1	0

（2進）

ビット

1バイト＝8ビット

1バイトで2^8=256通りの情報を表せます。

バイトの単位

バイトなどのコンピュータの単位では、2^{10}(=1024)ごとに単位が繰り上がります。

単位	読み方	意味
KB	キロバイト	1KB＝1024バイト
MB	メガバイト	1MB＝1024KB
GB	ギガバイト	1GB＝1024MB
TB	テラバイト	1TB＝1024GB

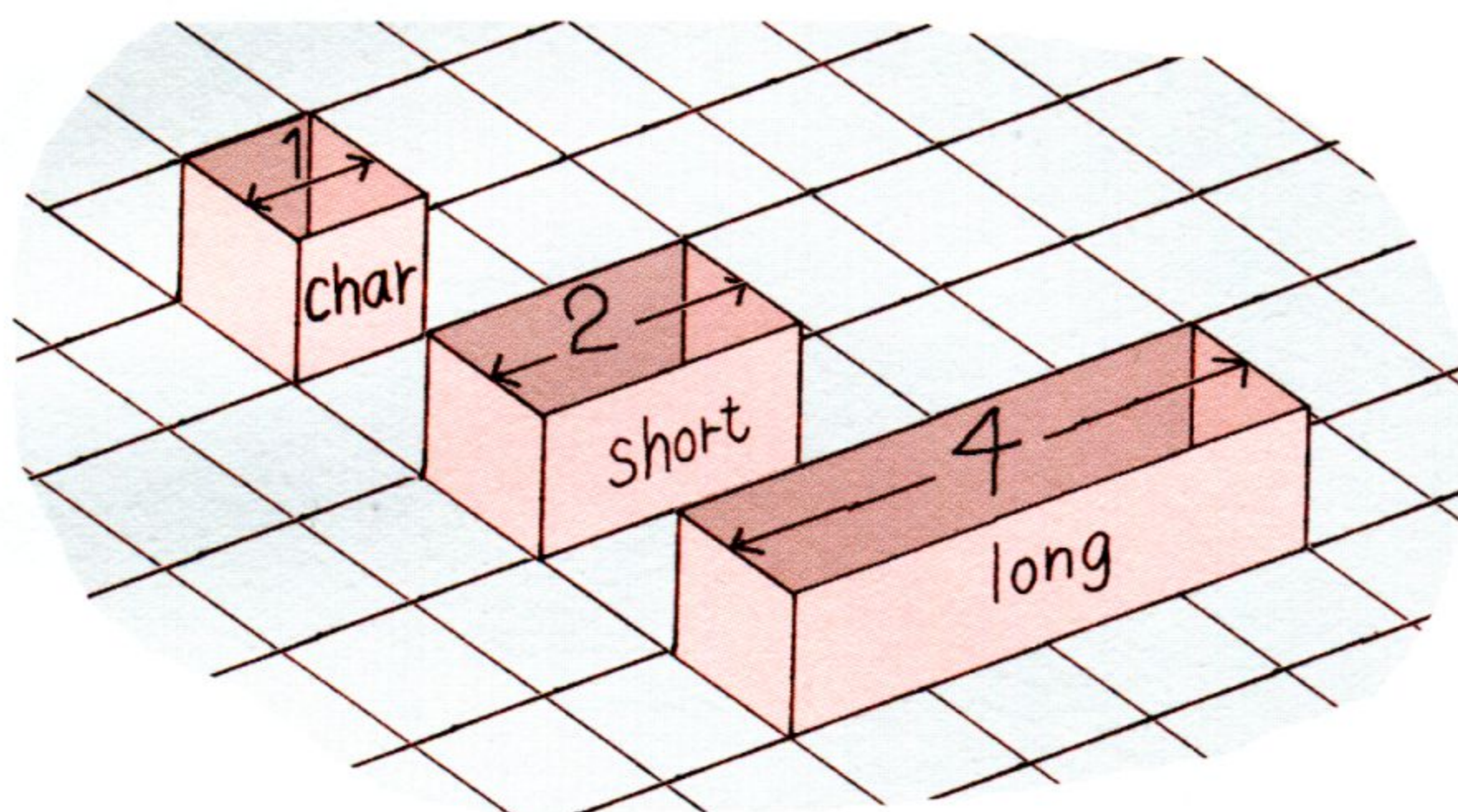
C サイズオブ sizeof演算子

sizeof演算子を使うと、変数や型がメモリ上に占めるバイト数を取得できます。

```
int n, m;
n = sizeof(short);
m = sizeof(n);
```

nはshort型のバイト数（＝2）になります。

mはint型のバイト数になります。（値は処理系により異なります。）

例

```
#include <stdio.h>

main()
{
    char c = 1;
    char s[10] = "Hello";

    printf("long型 = %dバイト¥n", sizeof(long));
    printf("char型変数 = %dバイト¥n", sizeof(c));
    printf("文字列変数 = %dバイト¥n", sizeof(s));
}
```

文字数ではなく、文字の入る箱のバイト数を取得します。

実行結果

```
long型 = 4バイト
char型変数 = 1バイト
文字列変数 = 10バイト
```

型の変換

C言語で計算を行う場合、変数の型がとても重要になります。ここでは型の変換について学びます。

計算の中の型変換

C言語では整数どうしで計算すると、その結果は整数になるという決まりがあります。だから、次のようにちょっと変なことが起きてしまいます。

3÷2の結果を求める(誤)

3 / 2 → **1**
整数　整数　整数

整数になるように自動的に小数点以下が切り捨てられてしまいます。

正しい値である1.5を算出するには、実数表記にして計算する必要があります。

3÷2の結果を求める(正)

3.0 / 2.0 → **1.5**
実数　実数　実数

例

```
#include <stdio.h>

main()
{
    printf("3÷2=%d¥n", 3/2);
    printf("3÷2=%f¥n", 3.0/2.0);
    printf("3÷2=%f¥n", 3.0/2);
    printf("3÷2=%f¥n", 3/2.0);
}
```

実行結果

```
3÷2=1
3÷2=1.500000
3÷2=1.500000
3÷2=1.500000
```

実数を含む計算の場合、整数は自動的に実数に変換されます。

整数どうしの演算では、一番範囲の広い型に変換されます。

```
short s = 536;
char c = 12;
int a = s + c;
```

charの範囲は-128〜127

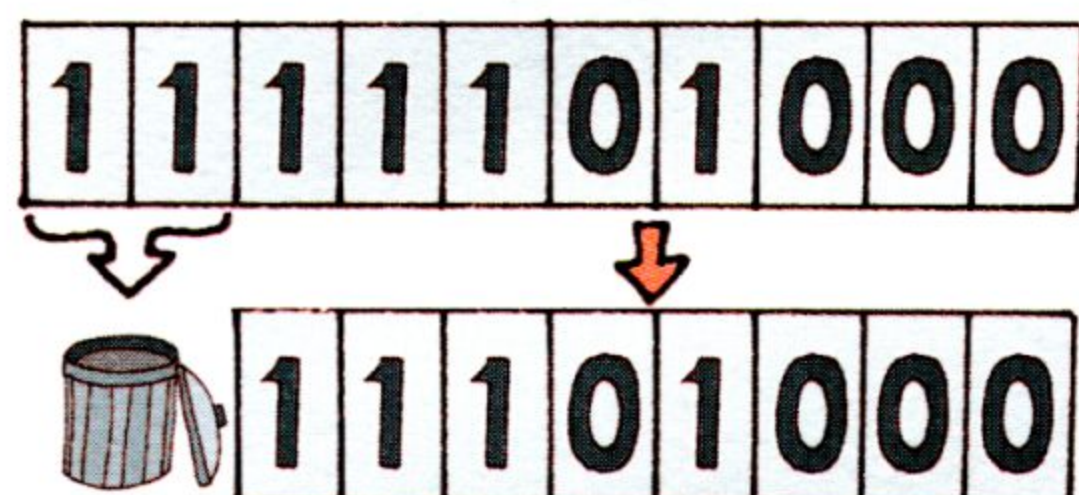
548になります。

異なる精度の型どうしの代入

値の範囲が異なる型の変数どうしで代入するときは、十分気をつける必要があります。

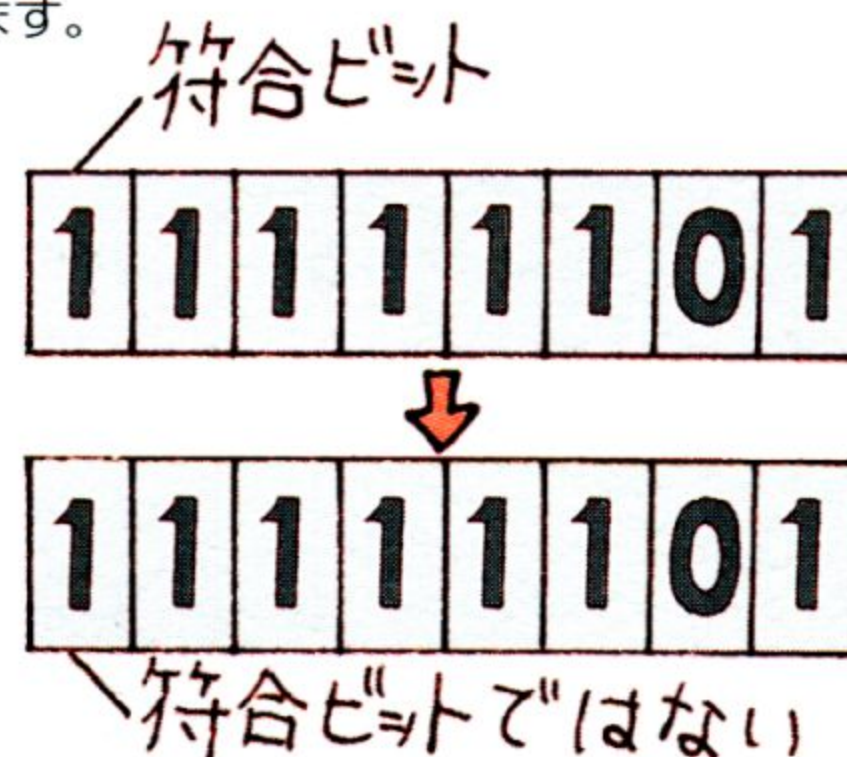
```
unsigned char c = 1000;
```

型の上限を超えた数を代入すると、あふれたビットは無視されます。
これを**オーバーフロー**といいます。

1111101000 → 11101000

cの値は232になってしまいます。

```
unsigned char c = -3;
```

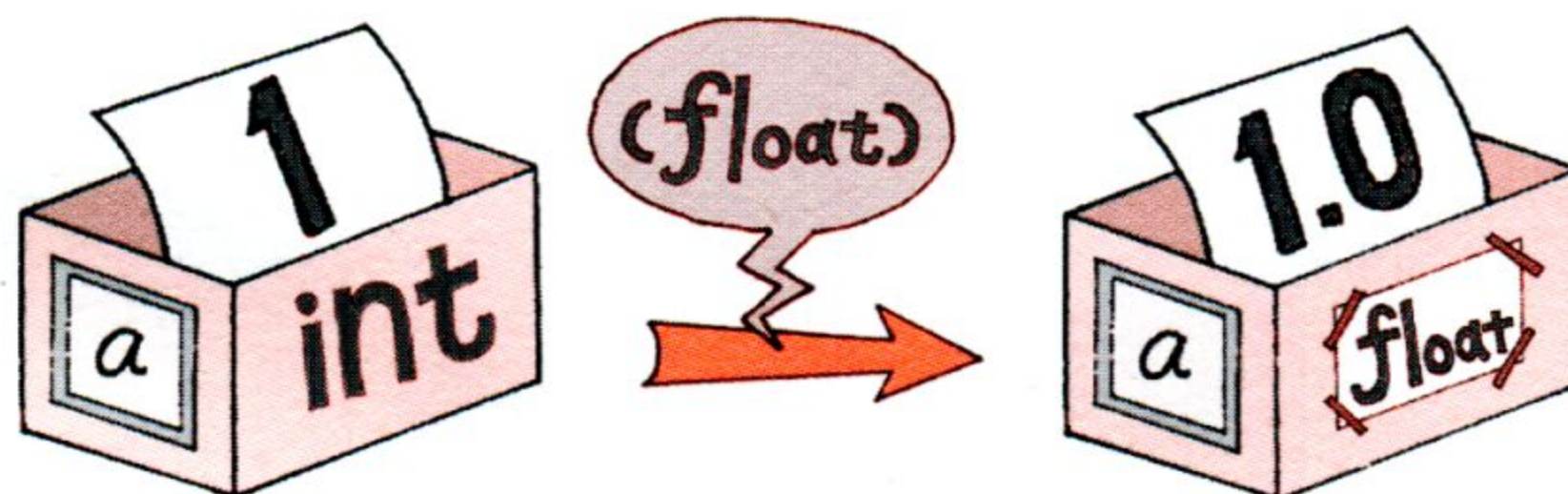
符号なしの変数に符号つきの値を入れると、ビット情報はそのままでも、符号なしと解釈します。

符号ビット
11111101 → 11111101
符号ビットではない

cの値は253になってしまいます。

キャスト演算子

「(int)」のように、型名を()でくくったものを値や変数の前に書くと、それらを特定の型に変換することができます。この操作を**型キャスト**といい、()を**キャスト演算子**といいます。

(float)
a int 1 → a float 1.0

例

```
#include <stdio.h>

main()
{
    printf("3÷2=%f¥n", 3/(float)2);
}
```

float型にキャスト

2で割った答えを算出する。

実行結果

```
3÷2=1.500000
```

演算の優先度

基本的な演算子が一通り登場したところで、演算子の優先順位を紹介しましょう。

演算子の優先順位

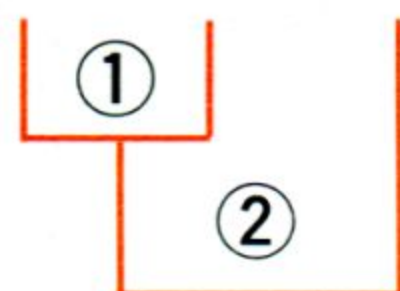
基本的に式は左から右へ計算していきますが、「×は＋よりも先に計算する」や「( )の中を先に計算する」など、演算には優先順位があります。式の中に複数の演算子が含まれる場合、C言語では次の優先順位に基づいて計算します。また、同じ順位の演算子が並んでいるとき、式の左右どちらから適用していくかも決まっています。

<table>
<tr><th>優先順位</th><th>演算子</th><th>同順位のときは
左から右（→）
右から左（←）
の順に式を実行</th></tr>
<tr><td>1</td><td>( )　[ ]　.（ピリオド、構造体のメンバの選択）　->
++（後置）　--（後置）</td><td>→</td></tr>
<tr><td>2</td><td>!　~　++（前置）　--（前置）　+（符号）　-（符号）
&（ポインタ）　*（ポインタ）　sizeof</td><td>←</td></tr>
<tr><td>3</td><td>キャスト演算子</td><td>←</td></tr>
<tr><td>4</td><td>*　/　%</td><td>→</td></tr>
<tr><td>5</td><td>+　-</td><td>→</td></tr>
<tr><td>6</td><td><<　>></td><td>→</td></tr>
<tr><td>7</td><td><　<=　>　>=</td><td>→</td></tr>
<tr><td>8</td><td>==　!=</td><td>→</td></tr>
<tr><td>9</td><td>&（ビット積）</td><td>→</td></tr>
<tr><td>10</td><td>^</td><td>→</td></tr>
<tr><td>11</td><td>|</td><td>→</td></tr>
<tr><td>12</td><td>&&</td><td>→</td></tr>
<tr><td>13</td><td>||</td><td>→</td></tr>
<tr><td>14</td><td>?:（三項演算子）</td><td>←</td></tr>
<tr><td>15</td><td>=　+=　-=　*=　/=　%=　&=　|=　^=　<<=　>>=</td><td>←</td></tr>
<tr><td>16</td><td>,（カンマ）</td><td>→</td></tr>
</table>

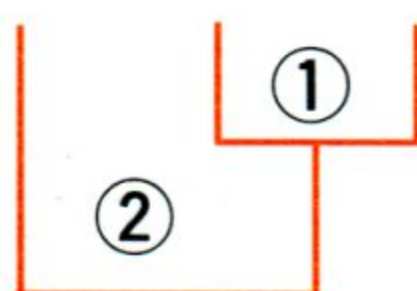
≫式の読み方

いろいろな演算子の優先順位を見てみましょう。

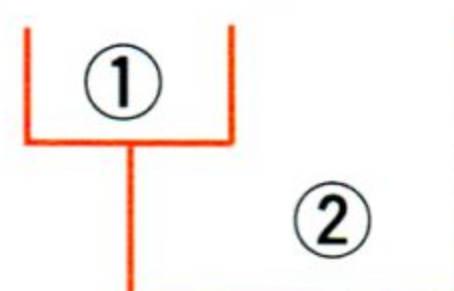
a + b - c

①(a + b) ②(① - c)

四則演算は同じ優先度なら左から計算します。

a + b * c

①(b * c) ②(a + ①)

＋や－よりも＊や/の方を先に計算します。

(a + b) * c

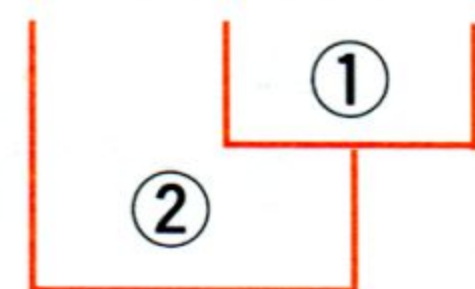
①(a + b) ②(① * c)

( )でくくると、その中を先に計算します。

a = b = c = 1

①(c = 1) ②(b = ①) ③(a = ②)

右の代入から先に実行します。a、b、cの値はいずれも1になります。

c = a == b

①(a == b) ②(c = ①)

aとbが等しければ1を、異なれば0をcに代入します。

例

```
#include <stdio.h>

main()
{
    printf("2×8－6÷2 ＝ %d¥n", 2*8-6/2);
    printf("2×(8－6)÷2 ＝ %d¥n", 2*(8-6)/2);
    printf("1－2＋3 ＝ %d¥n", 1-2+3);
    printf("1－(2＋3) ＝ %d¥n", 1-(2+3));
}
```

実行結果

```
2×8－6÷2 ＝ 13
2×(8－6)÷2 ＝ 2
1－2＋3 ＝ 2
1－(2＋3) ＝ -4
```

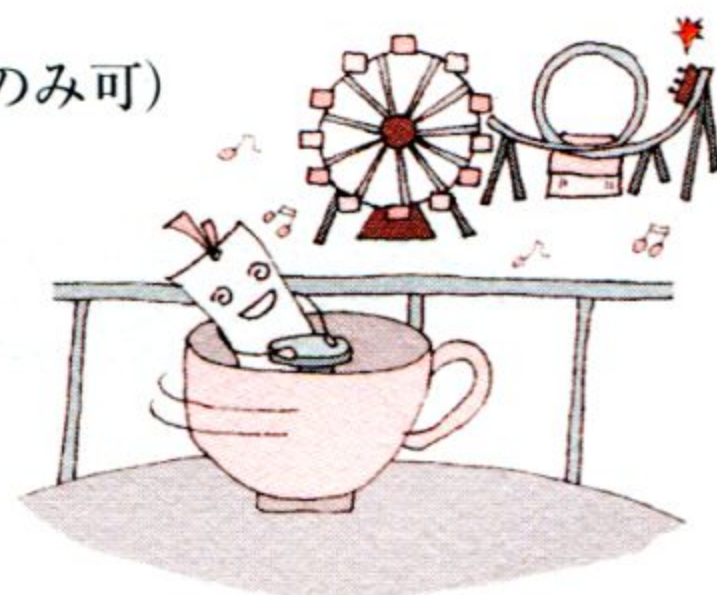
COLUMN コラム

～複雑な論理演算～

論理演算とは、いろいろな条件の組み合わせが、成立するかどうかを「真（true）」か「偽(false)」という値で導き出すものでした。とても難しいことのように考えがちですが、実はこうしたいろいろな条件をふまえて判断する作業は、私たちの日常生活の中にあふれています。たとえばお店で買い物をして「395円です」といわれたとします。そうしたら、まずあなたは財布の小銭入れを見て、どんな硬貨があるかを確認するでしょう。ぴったり395円があるかもしれないし、5円玉しかないかもしれません。もし、5円玉しかなければ今度は札入れを確認しますね。何気なく行っているこうした動作は、そのひとつひとつを取ってみると立派な判断作業といえます。

別の例で、具体的に論理演算と結びつけてみましょう。遊園地のアトラクションの中には、乗るために一定の条件をクリアしなければならないものがありますね。例えば次のようなものです。

1. 6歳以上（ただし、身長130cm以上であれば保護者同伴の場合のみ可）
2. 身長130cm以上
3. 心臓の弱い方はご遠慮ください

年齢をage、身長をheightとし、「健康であること」をhealth、「保護者同伴であること」をpgとすると、このアトラクションに乗れるための条件は次のようになります。わかりますか？

```
((age >= 6 && height >= 130) || (height >= 130 && pg)) && health
```

もうひとつ、うるう年かそうでないかの条件式をご紹介しましょう。

その年がうるう年になるためには以下の条件が成り立つ必要があります。

1. 西暦が4で割り切れる
2. ただし例外として、西暦が100で割り切れる年は除く
3. さらに例外として、西暦が400で割り切れる場合は含める

とても複雑な条件のように思えますが、これをC言語の式で表すと次のようになります。

(変数aを西暦とします)

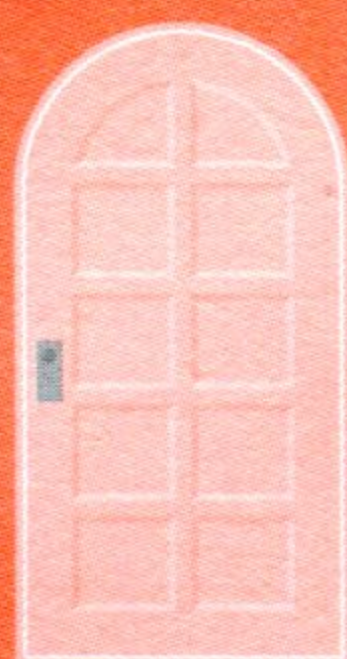
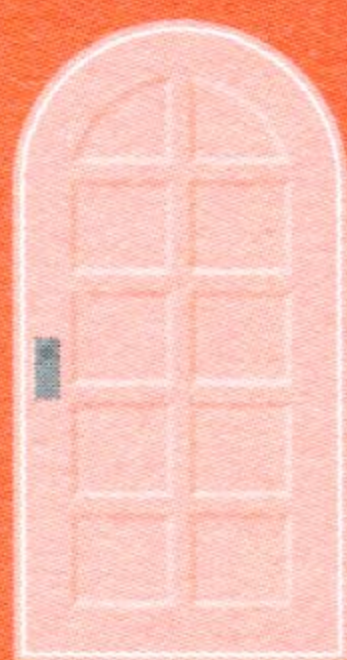
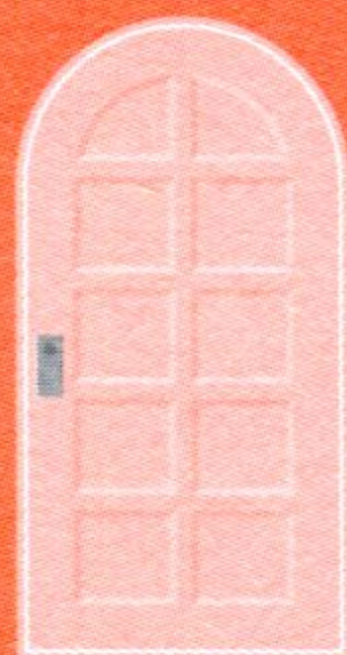
```
(a % 4 == 0 && a % 100 != 0) || a % 400 == 0
```

1の条件　2の条件　3の条件

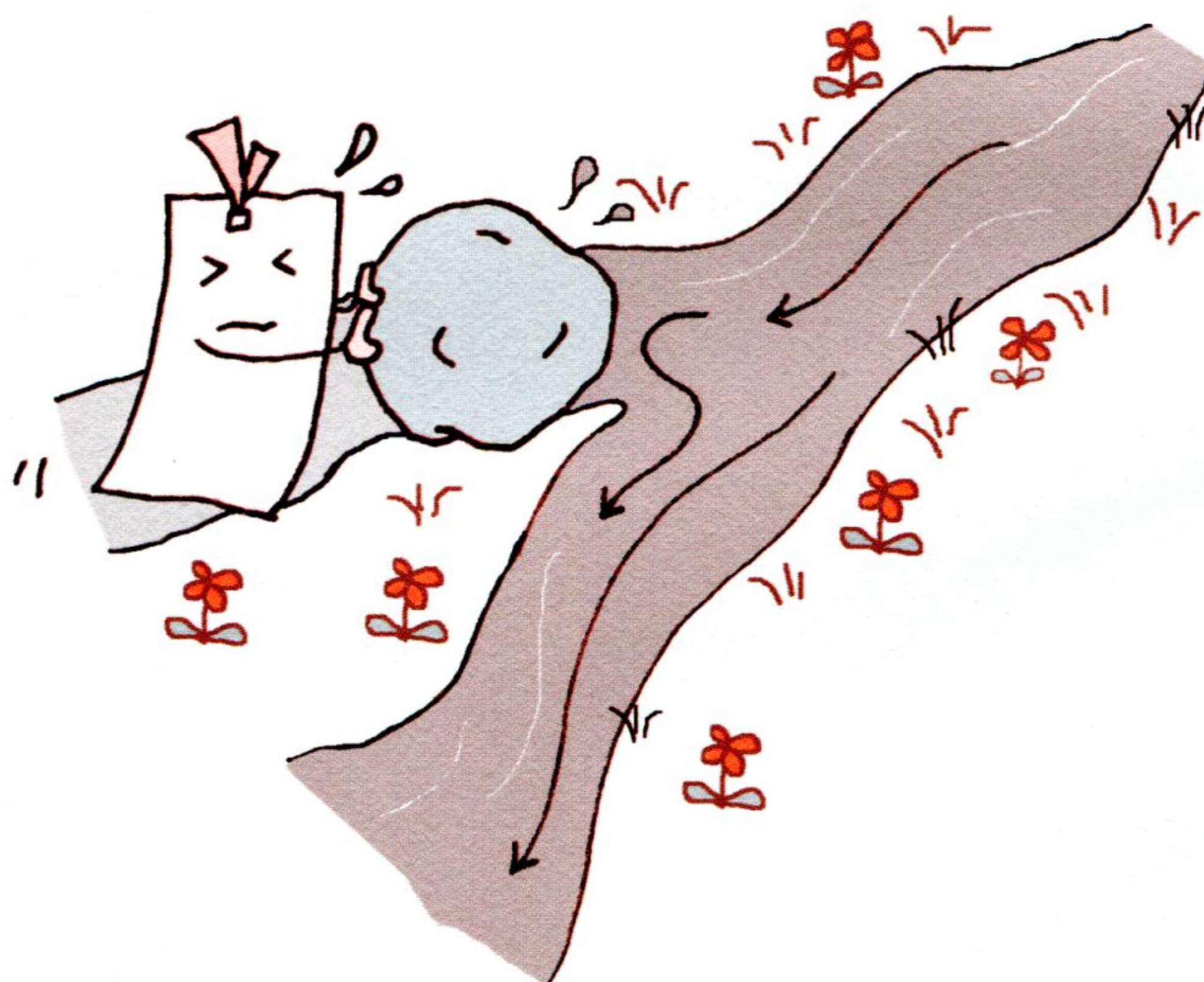
上の式の値が1（真）になればうるう年、0（偽）になればうるう年ではない、ということになります。

3 制御文

第3章

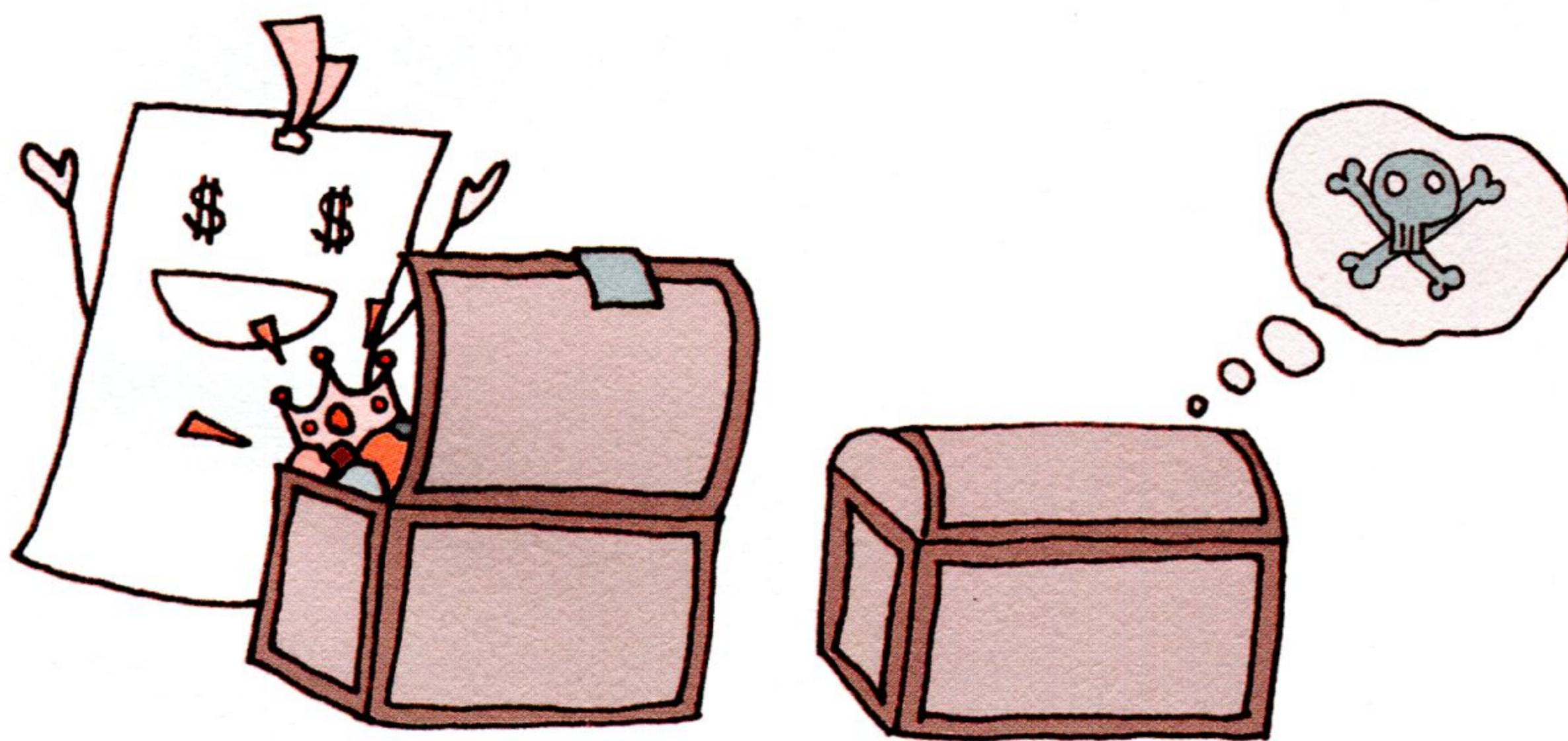
第3章は ここがkey！

Topics プログラムの流れを変えてみよう！

この章では実際にプログラミングをする上でよく使われる**制御文**について紹介します。制御文はプログラムの流れを必要に応じて変えたいときに使うものです。

プログラムは本来、水のように上から下に向かって流れていきますが、それでは単純な動作しか定義できません。状況によっては、「同じ処理を繰り返す」「演算結果によって処理を中止したい」ということもあるでしょう。そんなときに活躍するのが制御文です。制御文を使えばプログラムの流れを戻したり、せき止めたりすることも可能になります。

はじめに紹介するのは**if**(イフ)文です。これは英語の「if」という単語の意味のとおり、「もし～だったら…する」という、条件分岐を作る制御文です。つまり、条件が「成り立った場

合」と「成り立たなかった場合」の2通りのプログラムの流れを用意することができるのです。もちろん、if文を複数使用することにより2つ以上の流れを作ることも可能です。

次に登場するのが**for**（フォー）文と**while**（ホワイル）文です。これらはどちらも処理の「繰り返し」を行いたいときに使う制御文です。for文のページでは、たった4行のプログラムでコンピュータに九九の計算を行わせる例が出てきます。本編ではfor文とwhile文をそれぞれ別の項目に分け、詳しく紹介しています。

また、簡単に複数の分岐ができる**switch**（スウィッチ）文という制御文も紹介します。ロールプレイングゲームなどで、選択項目の中から選んだ項目によって、その後のゲームの流れが変わってくる場合などに使えそうです。

制御文を使えば、コンピュータに複雑な処理をさせることが可能になります。しかし、プログラムの流れを変えると**無限ループ**（永久に続く繰り返し）などいろいろ間違ったプログラムを書いてしまうケースも増えてきます。それぞれの制御文を正しく理解し、十分に気をつけてプログラミングするようにしましょう。

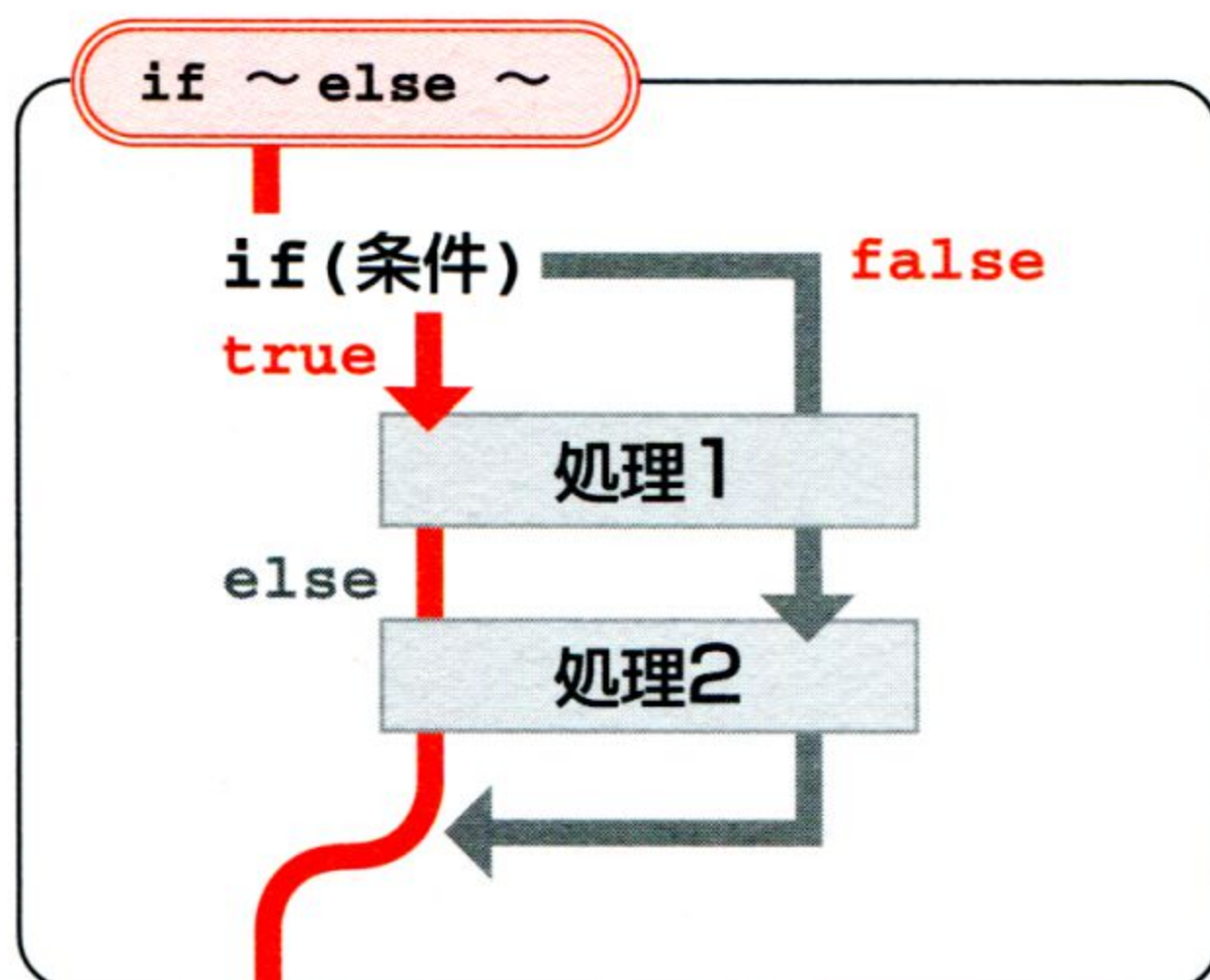
if文(1)

制御文のifは、英単語の「if（もし〜だったら）」と同じ意味です。C言語の制御文の中では、一番基本的なものです。

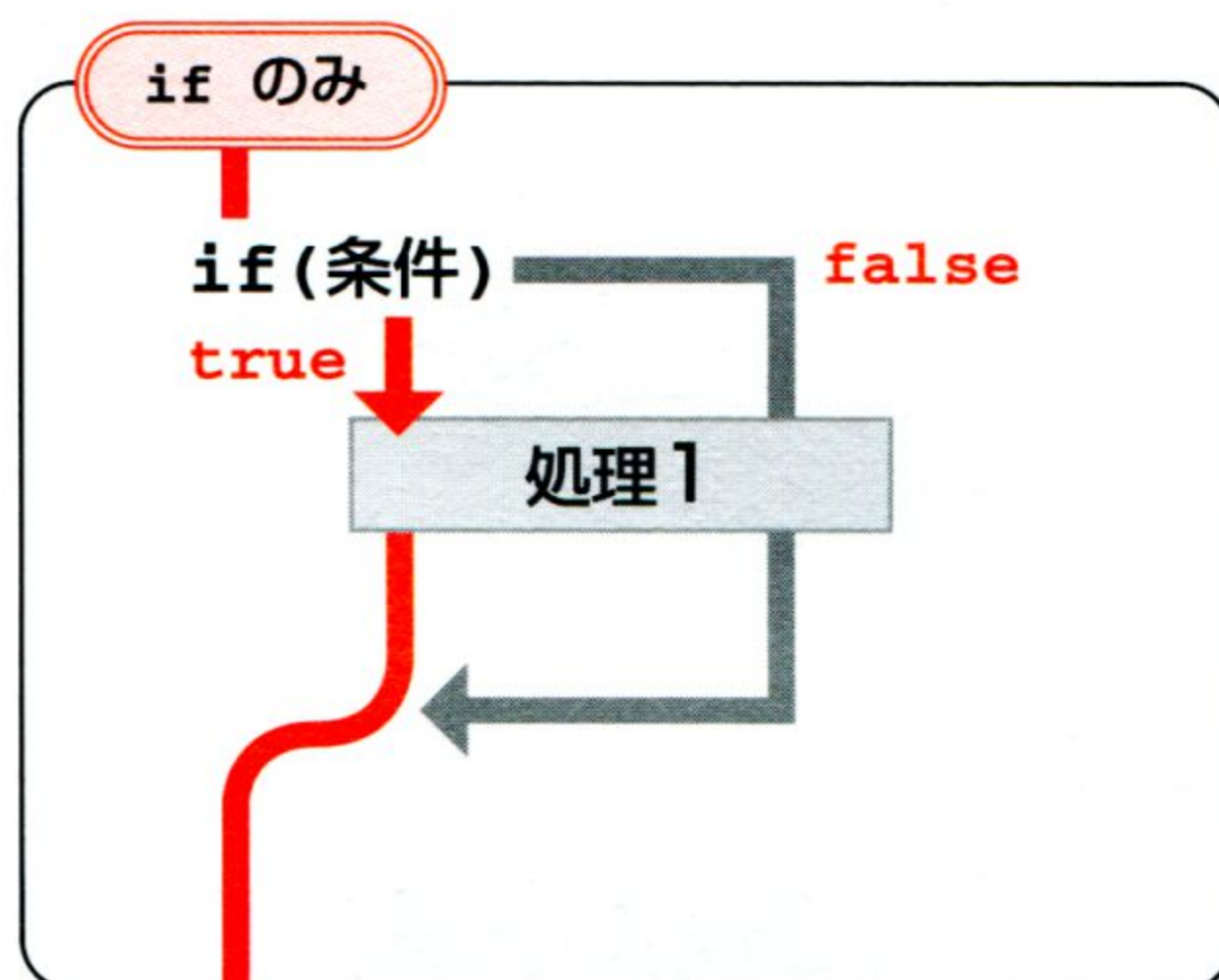
if文とは？

if文は条件によって処理を振り分けるときに使います。条件には比較演算子や論理演算子を使った条件式を指定します。

if 〜else 〜

if(条件) true → 処理1 / false → 処理2 (else)

条件が成り立つとき（true）は処理1を、成り立たないとき（false）は処理2を行います。

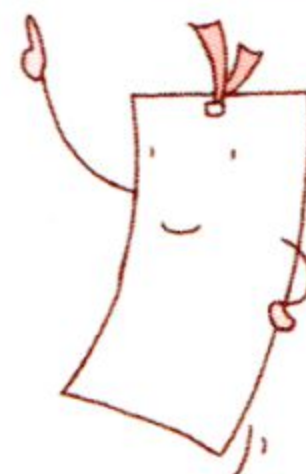
if のみ

if(条件) true → 処理1 / false

条件が成り立つときは処理1を行います。成り立たないときは何もしません。

例

```
#include <stdio.h>

main()
{
    int a = 5;

    if(a%2 == 0)
        printf("%dは偶数です。¥n", a);
    else
        printf("%dは奇数です。¥n", a);
}
```

実行結果

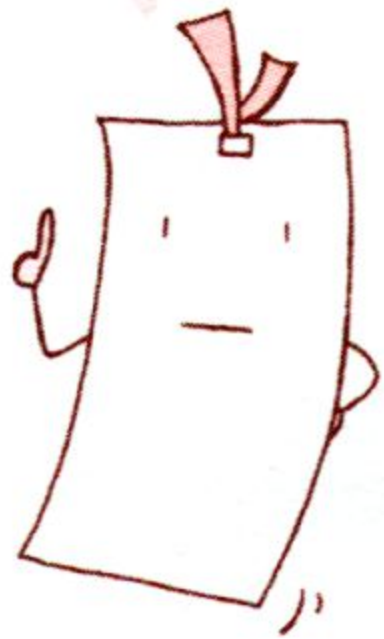
```
5は奇数です。
```

5÷2の余りは1なので、else以下の処理を実行します。

≫ブロック

左ページの処理1と処理2のところには、基本的に1つの文しか書けないことになっています。複数の処理を行いたい場合は、それらの文全体を中カッコ{ }でくくって1つと見なします。これを**ブロック**といいます。

ブロックの中はタブで字下げしたほうが見やすくなります。

```
if(条件式)
{
        XXXXXXXXXX      ┐
        XXXXXXXXXX      ┘ ブロック
}
else
{
        XXXXXXXXXX      ┐
        XXXXXXXXXX      ┘ ブロック
}
```

字下げ

スペース節約のためこのように書くことも多いです。

```
if(条件式) {
        XXXXXXXXXX
        XXXXXXXXXX
} else {
        XXXXXXXXXX
        XXXXXXXXXX
}
```

例

```
#include <stdio.h>

main()
{
    int s = 65;

    printf("あなたの点数は%d点です。¥n", s);

    if(s < 70)
    {
        printf("平均まであと%d点。¥n", 70-s);
        printf("がんばりましょう！¥n");
    }
    else
    {
        printf("よくできました！¥n");
    }
}
```

ブロック

ブロックは不要ですが、あってもかまいません。

実行結果

```
あなたの点数は65点です。
平均まであと5点。
がんばりましょう！
```

if文(2)

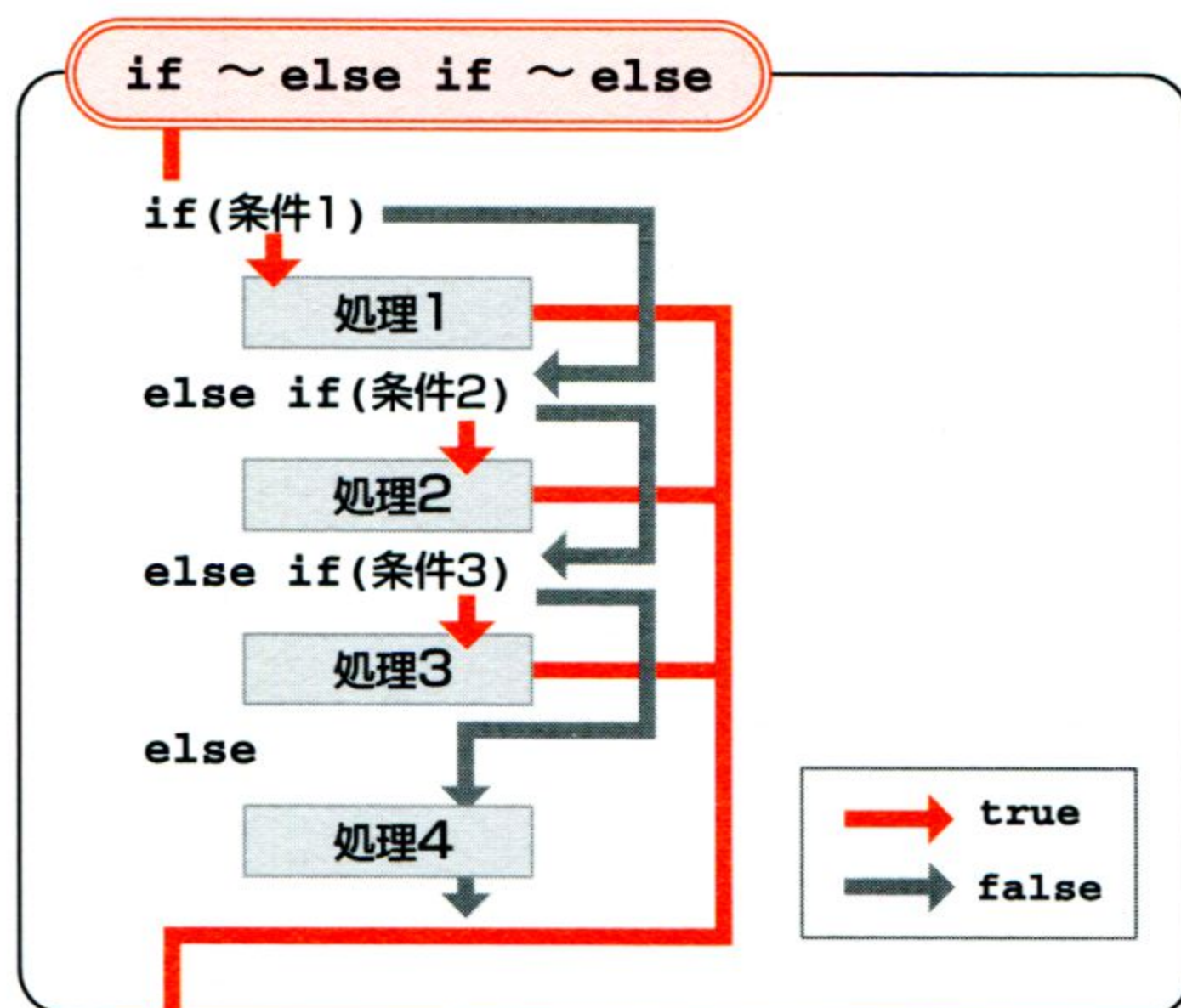
複雑な構造を持つプログラムで使われる、if文の応用を学びましょう。

連続したif文

複数の条件のどれにあてはまるかによって、それぞれ違う処理を行いたいときはif文を組み合わせて使います。

if ～else if ～else

if(条件1)
処理1
else if(条件2)
処理2
else if(条件3)
処理3
else
処理4

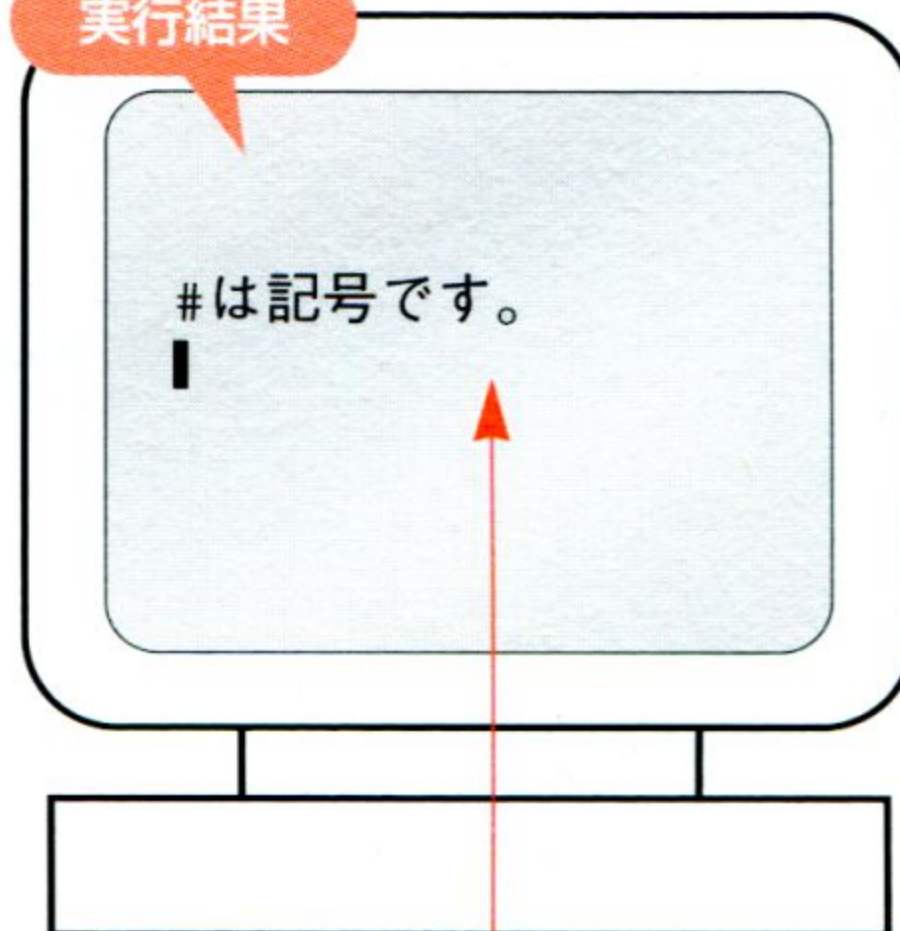
true
false

条件1が成立 → 処理1を実行
条件2が成立 → 処理2を実行
条件3が成立 → 処理3を実行
どれも成立しない → 処理4を実行

実行する処理はどれか1つです。

例

```
#include <stdio.h>

main()
{
    char c = '#';

    printf("%cは", c);

    if('0' <= c && c <= '9')
        printf("数字です。¥n");
    else if('a' <= c && c <= 'z')
        printf("小文字です。¥n");
    else if('A' <= c && c <= 'Z')
        printf("大文字です。¥n");
    else
        printf("記号です。¥n");
}
```

実行結果

```
#は記号です。
```

条件のどれにもあてはまらないので、else以下を実行します。

C 入れ子になったif文

if文をはじめとする制御文では、処理の中にさらに制御文を含めることができます。このような入れ子のことを**ネスト**といいます。

正しく字下げしておけば見やすくなります。

```
if(条件1)
{
        if(条件2)
                XXXXXXXXXX
        else
                XXXXXXXXXX
}
else
        XXXXXXXXXX
```

1階層目

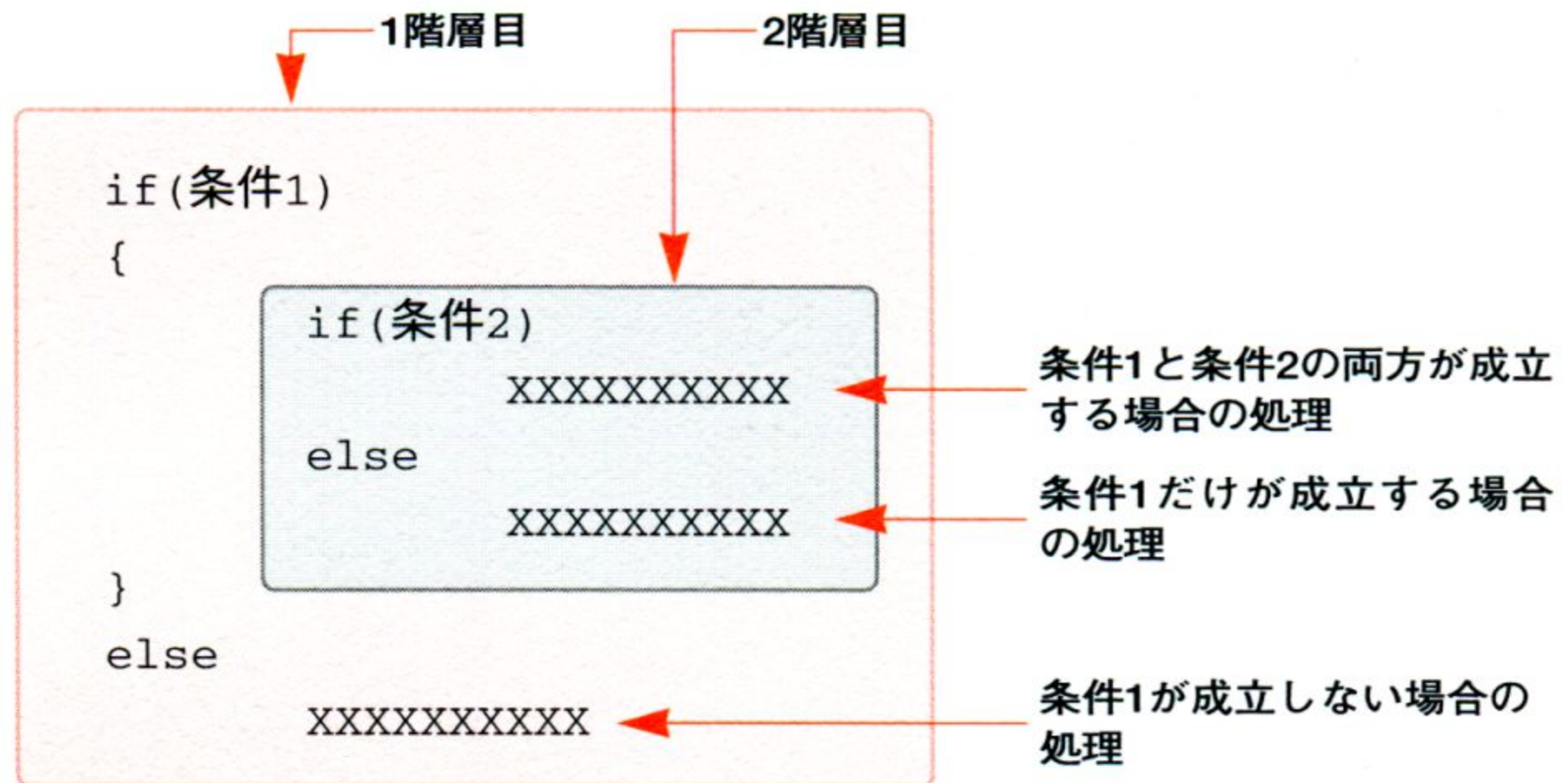
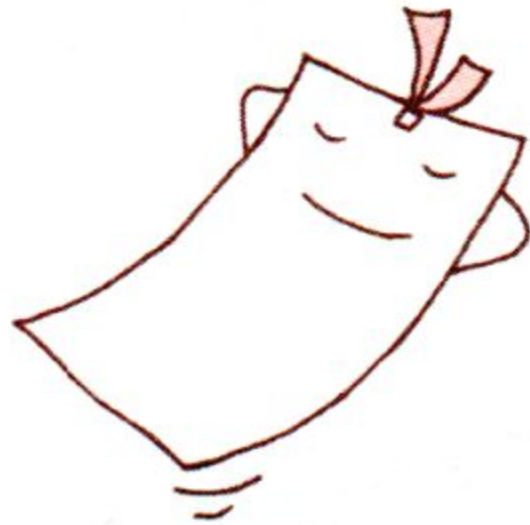
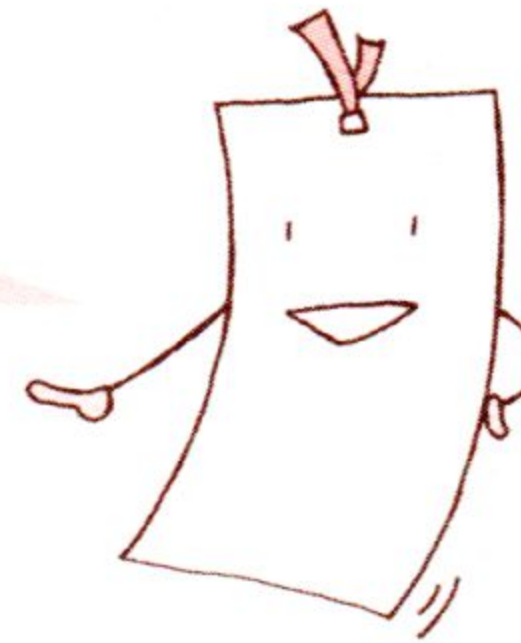
2階層目

条件1と条件2の両方が成立する場合の処理

条件1だけが成立する場合の処理

条件1が成立しない場合の処理

例

条件が成立した場合の判断でif文をネストしています。

```
#include <stdio.h>

main()
{
    int a = 90;

    if(a > 80)
    {
        if(a == 100)
            printf("満点です。¥n");
        else
            printf("もう少しです。¥n");
    }
    else
        printf("がんばりましょう。¥n");
}
```

実行結果

```
もう少しです。
```

for文

プログラムでは、同じような処理を繰り返すことがよくありますね。そんなときは、for文を使います。

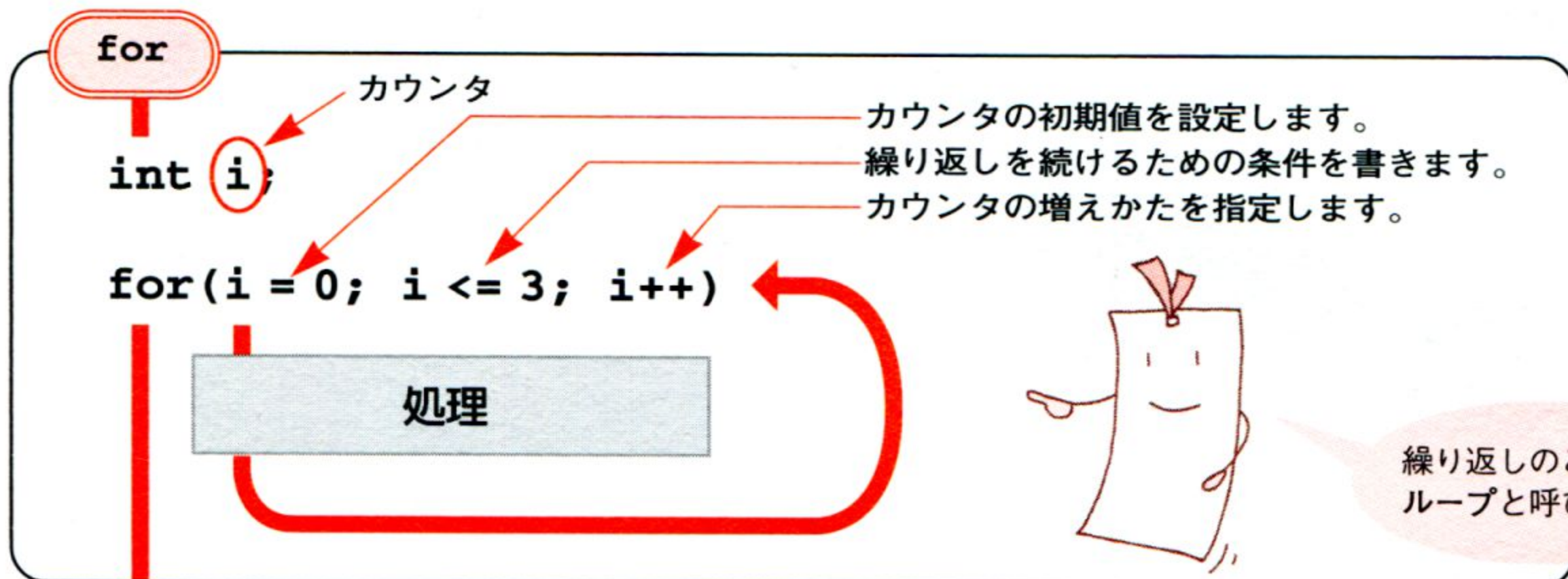
for文とは？

for文は、繰り返し処理を効率よく行うための制御文です。ふつうはカウンタを用意して、その値によって何回繰り返すかを決めます。

for

```
int i;

for(i = 0; i <= 3; i++)
    処理
```

- カウンタ（i）
- カウンタの初期値を設定します。
- 繰り返しを続けるための条件を書きます。
- カウンタの増えかたを指定します。

繰り返しのことをループと呼びます。

iの初期値を0として、1つずつ値を増やしていき、3以下であるあいだ処理を繰り返し実行します。

例

```
#include <stdio.h>

main()
{
    int i;
    for(i = 1; i < 4; i++)
        printf("こんにちは%d¥n", i);
}
```

処理の順序

1. 変数iに1を代入
2. "こんにちは1"を表示
3. i++を実行（i = 2）
4. i < 4なので、繰り返す
5. "こんにちは2"を表示
6. i++を実行（i = 3）
7. i < 4なので、繰り返す
8. "こんにちは3"を表示
9. i++を実行（i = 4）
10. i < 4ではないのでループ終了

実行結果

```
こんにちは1
こんにちは2
こんにちは3
```

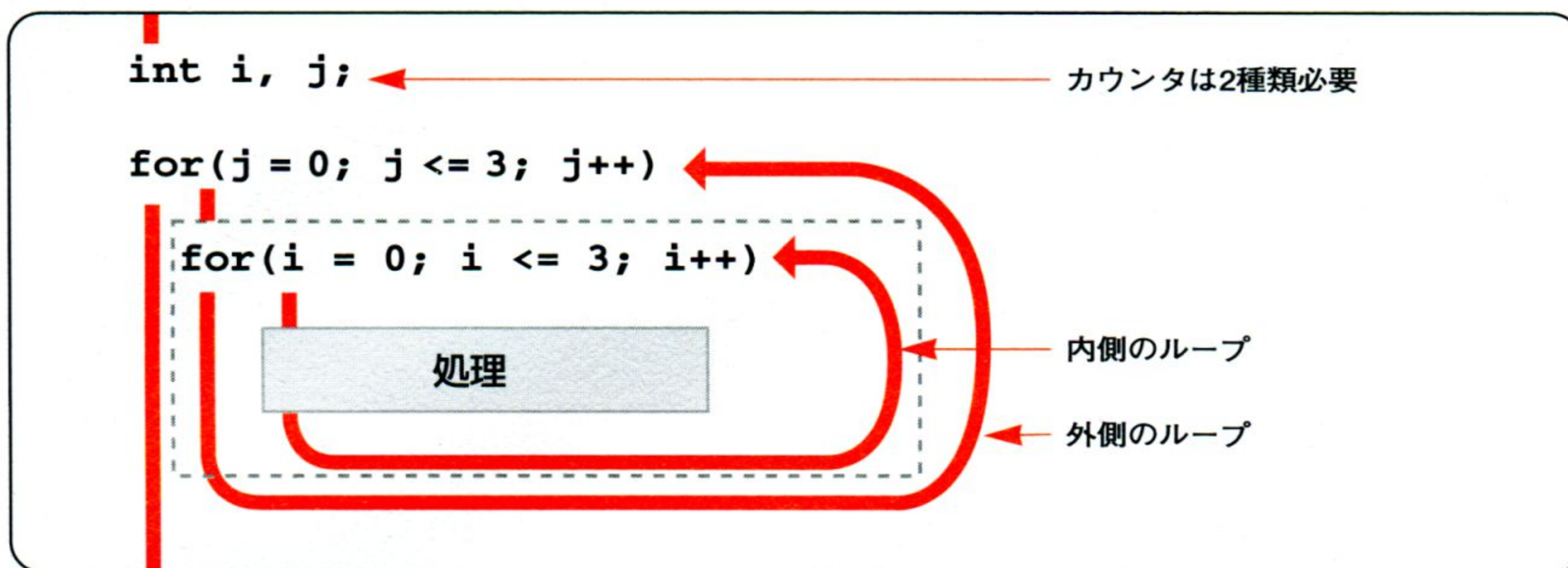
≫2重ループ

for文を2つ使って、繰り返しの中に繰り返しを書くこともできます。これを**2重ループ**といいます。

```
int i, j;

for(j = 0; j <= 3; j++)
    for(i = 0; i <= 3; i++)
        処理
```

カウンタは2種類必要

内側のループ

外側のループ

2重ループでは、値の移り変わりは次のようになります。

```
int i, j;

for(j = 1; j <= 2; j++)
    for(i = 1; i <= 3; i++)
        printf("%d¥n", i-j);
```

j	i	i-j
	1	0
1	2	1
	3	2
	1	-1
2	2	0
	3	1

処理の順序

例

```
#include <stdio.h>

main()
{
    int i, j;

    for(j = 1; j <= 9; j++)
        for(i = 1; i <= 9; i++)
            printf("%d×%d=%d¥n",j, i, j*i);
}
```

九九の計算をすべて表示します。

実行結果

```
1×1=1
1×2=2
  ⋮
9×8=72
9×9=81
```

while文

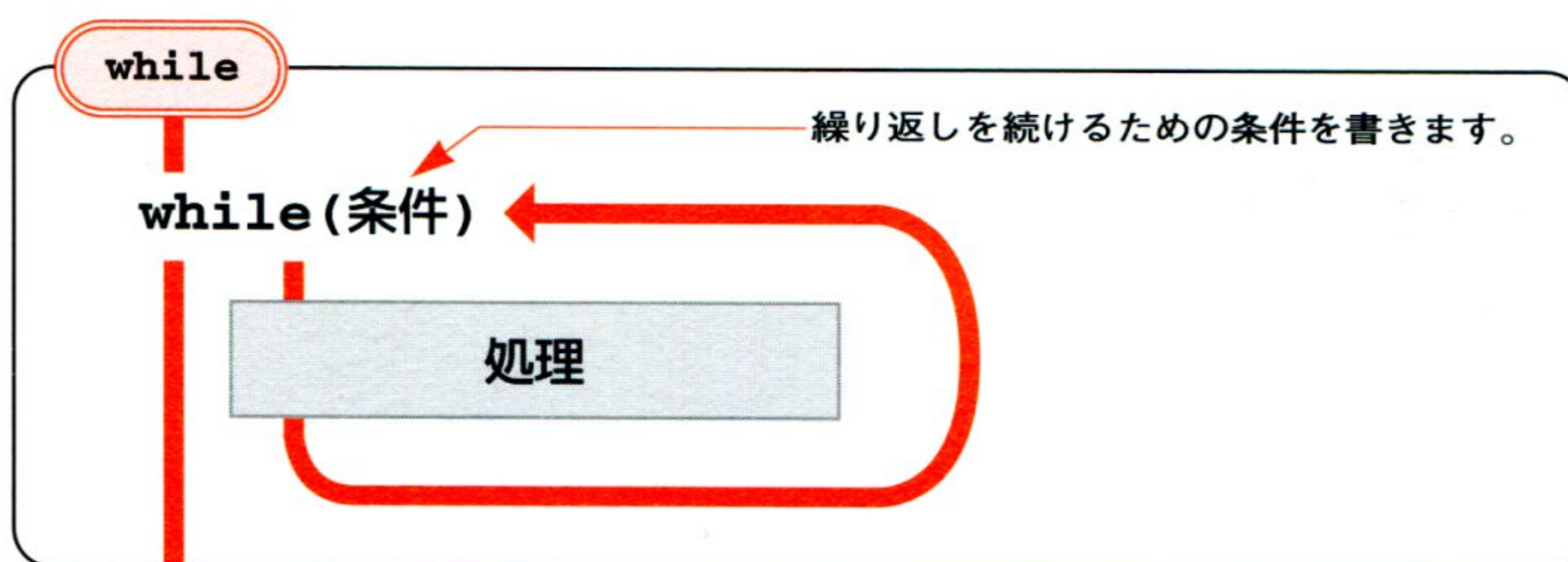
繰り返しを行う回数があらかじめ決まっていないときは、while文を使います。

while文とは？

while文は、ある条件が成り立っているあいだだけ、繰り返しを実行する制御文です。for文と異なるのは、カウンタにあたるものがないことです。主にキーボードからの入力など、繰り返す回数がわからないときに使います。

while

繰り返しを続けるための条件を書きます。

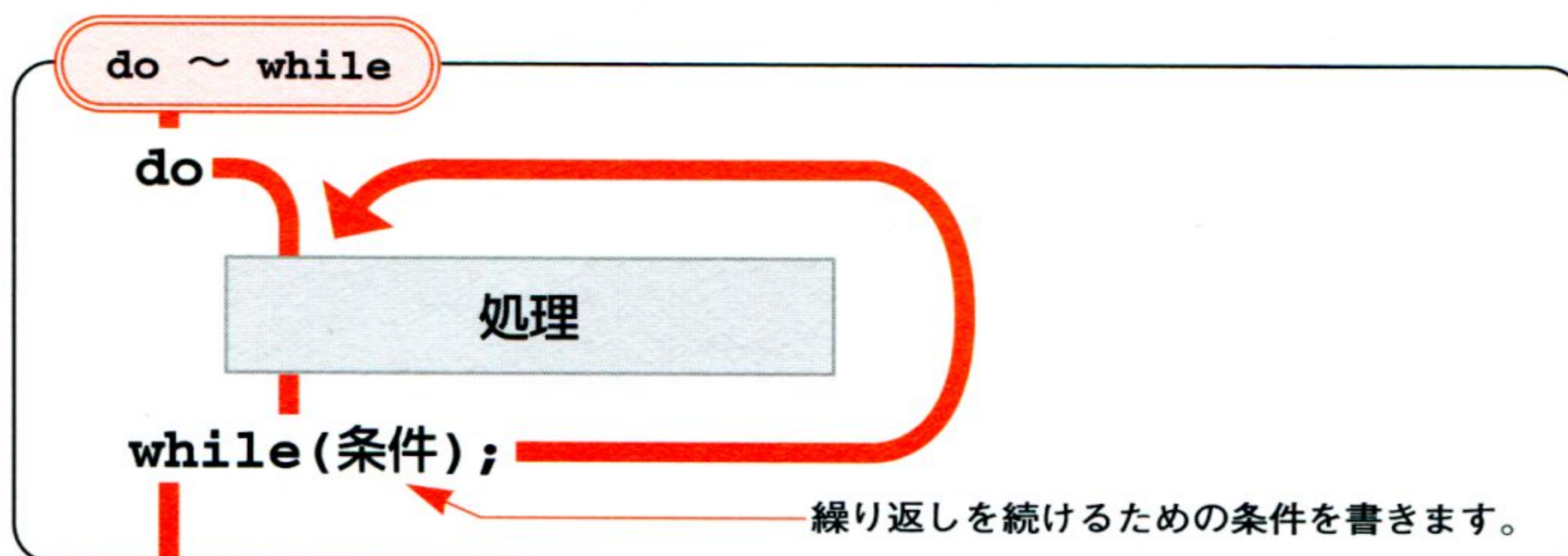
```
while(条件)
    処理
```

条件が成立する限り処理を繰り返します。

do ～ while文

do ～ while文も、while文と同じように繰り返しを行う制御文です。while文では処理よりも先に条件を評価するため、最初の回で条件が成立しなければ繰り返しを1度も実行しないことがあるのに対し、do ～ while文では条件を下に書くため、必ず1度は処理を実行します。

do ～ while

```
do
    処理
while(条件);
```

繰り返しを続けるための条件を書きます。

条件が成立する限り処理を繰り返します（必ず1度は実行します）。

例

```
#include <stdio.h>

main()
{
    char a;
    do {
        a = getchar();
        printf("%c", a);
    } while(a != 'e');
}
```

getchar()関数（ゲットチャー）
キーボードから入力された半角文字1つを得ます。

キーボードからeを入力するまで表示します。

do～whileなので「e」を表示してからループが終了します。

実行結果

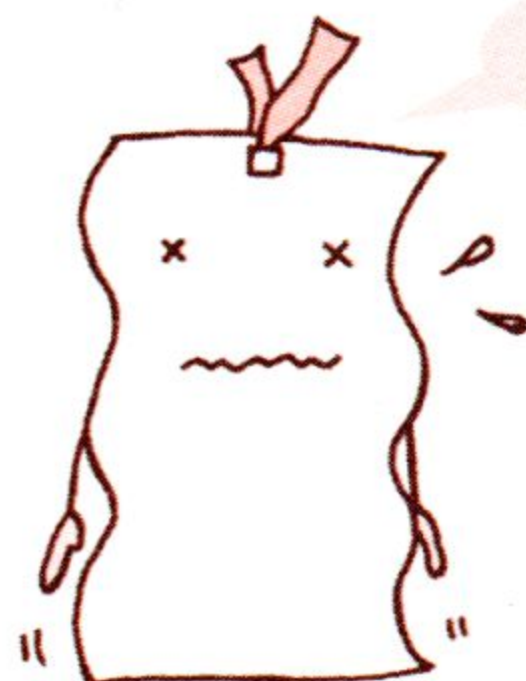
```
two ↵
two
three ↵
thre
```

※太字はキーボードから入力した文字

C 無限ループに注意

whileなどの繰り返し制御文では、常に成立するような条件を誤って指定してしまうと、処理を永久に繰り返してしまいます。これを無限ループといい、プログラムの**バグ**（不具合）の1つです。
条件と繰り返し処理の内容に注意して、無限ループにならないようにしましょう。

```
int a = 0;

while(a < 5)
{
    printf("%d¥n", a);
    a == a + 1;
}
```

注意
a = a+1; としてaを増やすところを、間違えて書いてしまいました。これではaの値は変わらないので、無限ループになってしまいます。

ループから出られません。

ループの中断

繰り返し処理などで流れを変えるときに使う制御文break(ブレイク)とcontinue(コンティニュー)を紹介します。

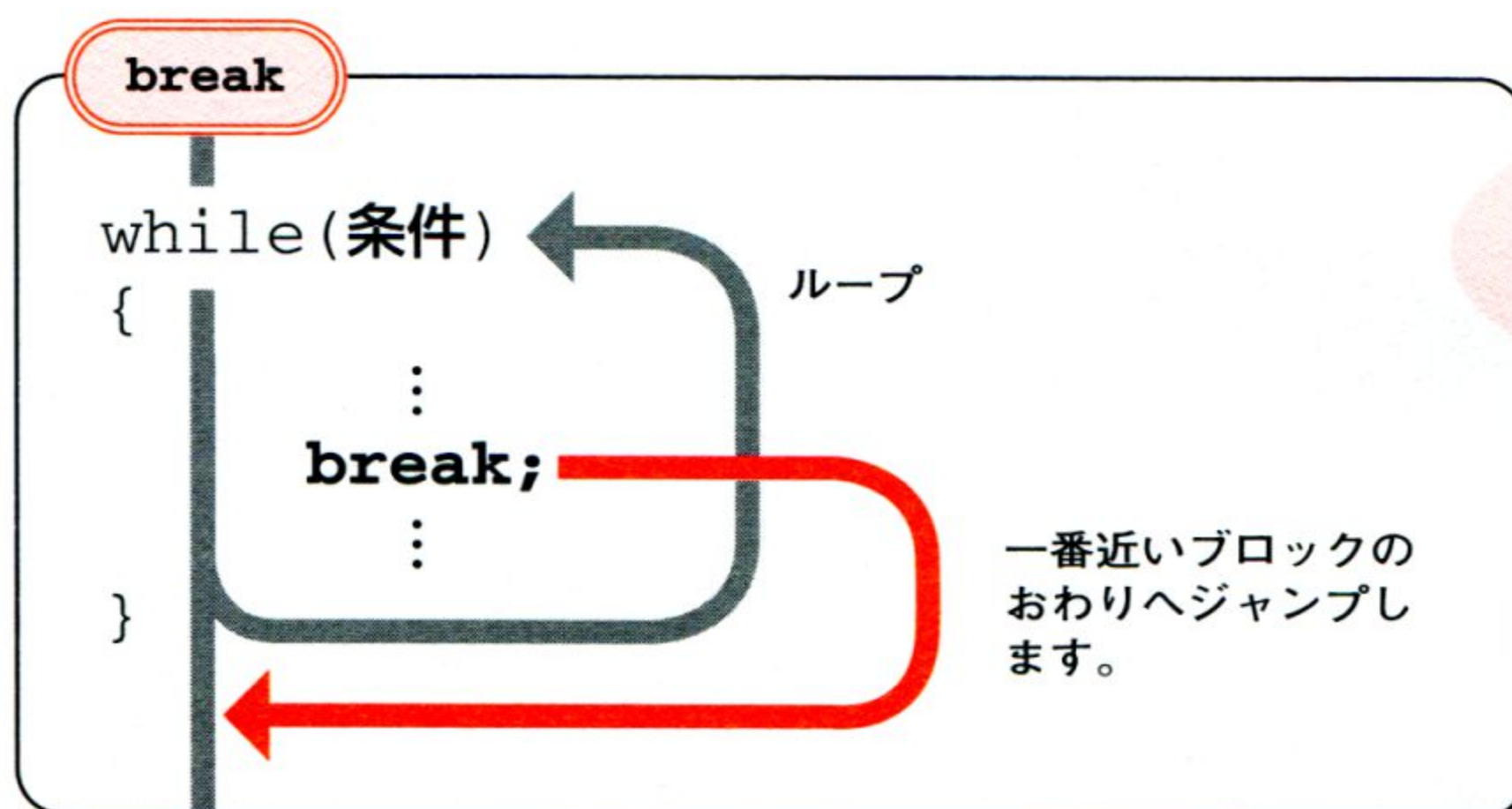
繰り返しを中断する

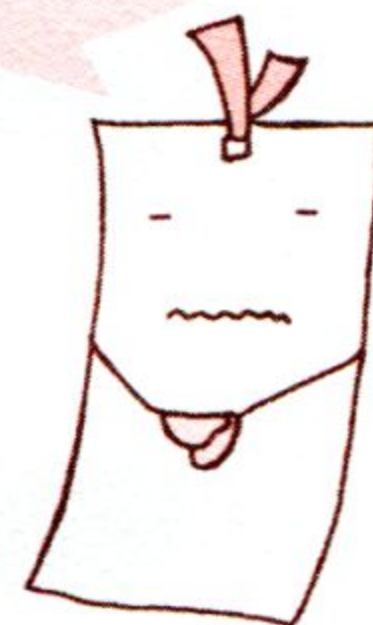
for文やwhile文などの繰り返しを途中で中断するには**break**文を使います。プログラム実行中にbreak文を見つけると、一番近いブロックのおわりにジャンプします。

break

```
while(条件)
{
      :
   break;
      :
}
```

ループ

一番近いブロックのおわりへジャンプします。

break文は複数のブロックを通過することはできません。

例

```
#include <stdio.h>

main()
{
    int a, b = 1;
    for(a = 0; a < 5; a++)
    {
        if(a+b >= 3)
            break;
        printf("%d+%d=%d¥n", a, b, a+b);
    }
}
```

a+bの値が3以上になったらループを終了します。

実行結果

```
0+1=1
1+1=2
```

2+1は3になるのでループを終了します。

C 繰り返しの次の回に移る

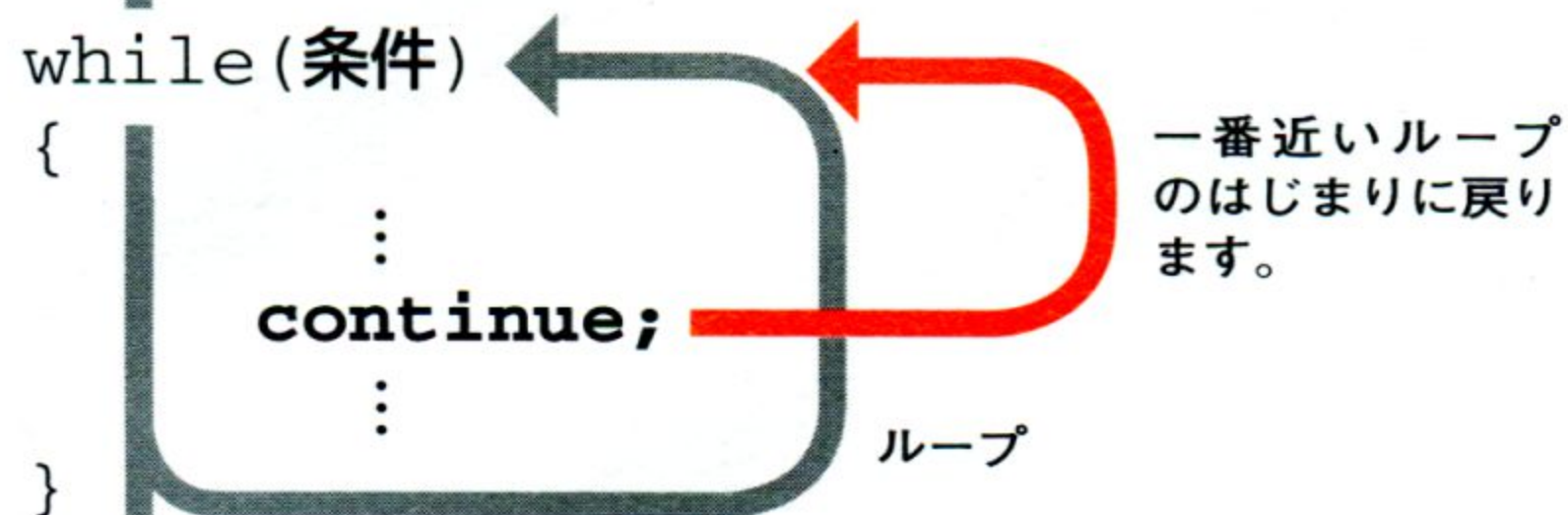
実行中のループ処理を中断するbreak文に対し、**continue**文は、繰り返しのその回の処理を中断し、次の回の最初から実行するという働きをします。

continue

```
while(条件)
{
      :
   continue;
      :
}
```

一番近いループのはじまりに戻ります。

ループ

例

```
#include <stdio.h>

main()
{
    int a, b = 1;
    for(a = 1; a < 5; a++)
    {
        if(a+b == 3)
            continue;
        printf("%d+%d=%d¥n", a, b, a+b);
    }
}
```

a+b=3のときは、ループのはじまりに戻ります。

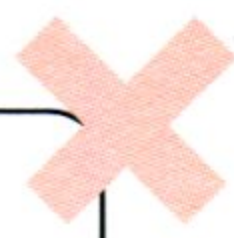
実行結果

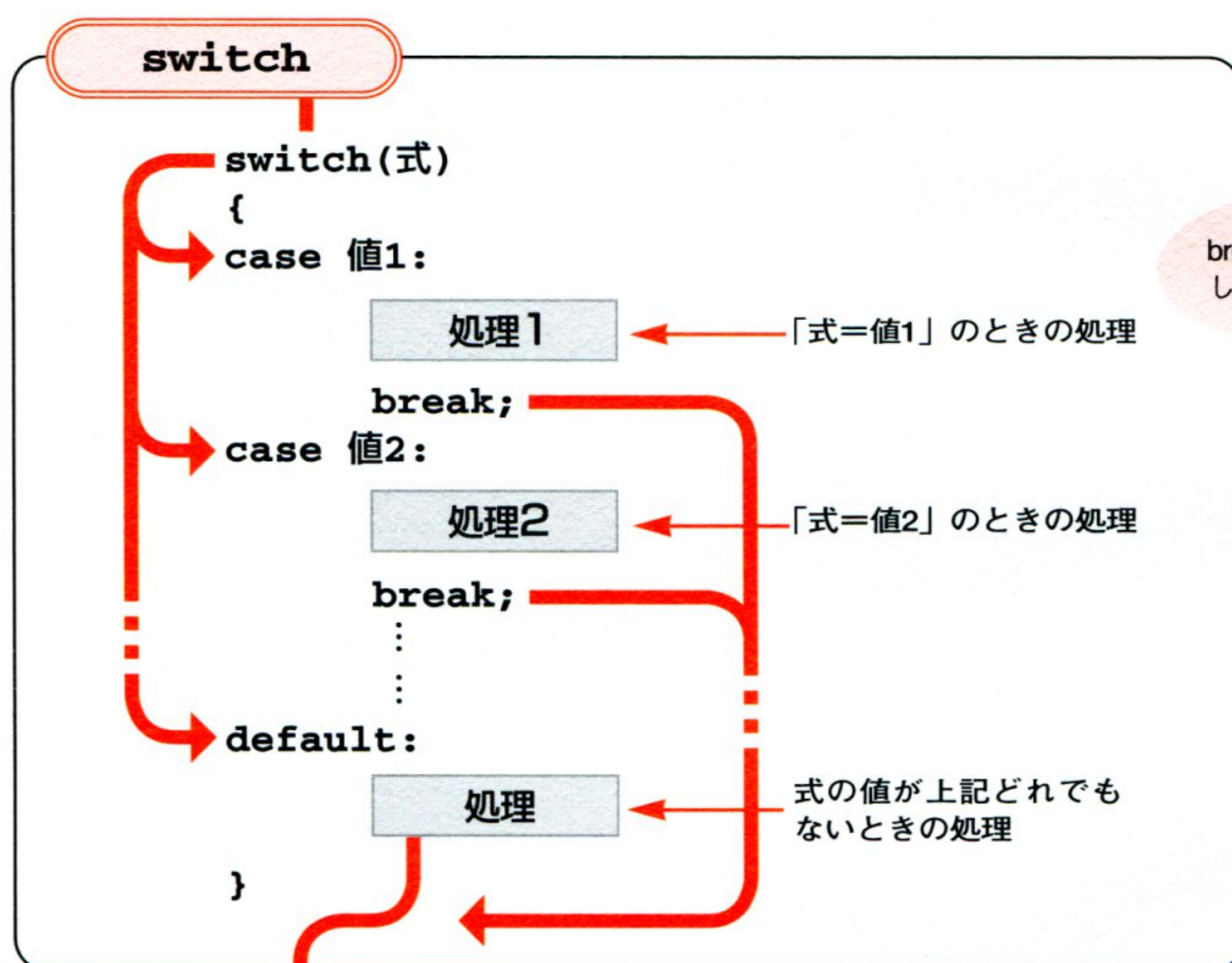
```
1+1=2
3+1=4
4+1=5
```

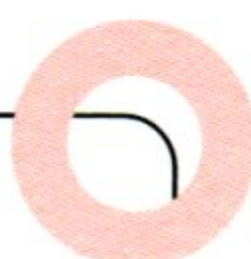
2+1は3になるので表示せず、次の回に進みます。

switch文

switch文を使うと、多くの選択肢を持つ分岐処理をスマートに記述できます。

処理を選択する

switch文は、複数の**case**（ケース）という選択肢の中から、式の値に合うものを選び、その処理を実行します。式の値がcaseのどれにもあてはまらないときは**default**（デフォルト）に進みます。各選択肢の最後にはbreak文を記述し、選択した処理のみを行うようにします。

```
switch(式)
{
case 値1:
        処理1        ←「式＝値1」のときの処理
        break;
case 値2:
        処理2        ←「式＝値2」のときの処理
        break;
        ⋮
default:
        処理         ←式の値が上記どれでもないときの処理
}
```


式の値によって異なる処理を選択し、実行します。

ただし、上の「式」には、値が数値であるものだけ使えます。それ以外のときは、代わりに「if～else if～else」を使ってください。

```
char s[16];

switch(s)
{
case "Hello":
    printf("Hello");
    break;
     ⋮
}
```

```
char s[16];

if(strcmp(s,"Hello") == 0)
    printf("Hello");
else if(・・・
    ⋮
    ⋮
```

例

```
#include <stdio.h>

main()
{
    char a;

    printf("1～3で好きな数字を入力してください¥n");
    a = getchar();

    switch(a)
    {
    case '1':
        printf("中吉¥n");
        break;
    case '2':
        printf("大吉¥n");
        break;
    case '3':
        printf("小吉¥n");
        break;
    default:
        printf("入力が間違っています¥n");
    }
}
```

文字はASCIIコードと同じなので、caseで使えます。

実行結果

```
1~3で好きな数字を入力してください
2 ↵
大吉
```

結果を表示

※太字はキーボードから入力した文字

サンプルプログラム

■ワードカウンタを作る

キーボードから入力した英文中の単語（ワード：スペースやピリオドで区切られた言葉のかたまり）の数を数えます。スペースがいくつか続くことを考慮する必要があります。[Enter] キーのみ入力するとプログラムを終了します。

ソースコード

```
#include <stdio.h>

main()
{
      char c = '¥0';          /* キーボードから入力した1文字 */
      char prevletter;        /* 以前の文字を取っておきます */
      int wordnum;            /* 単語の文字数 */
      int word_in;            /* 単語に入っていればtrue */

      while(1)
      {
            wordnum = 0;
            word_in = 1;
            prevletter = '¥0';
            printf("文字列を入力してください：");
            while(1)
            {
                  c = getchar();
                  if(c == '¥n')
                  {
                        if(word_in)
                              wordnum++;
                        break;
                  }
                  prevletter = c;
                  if(c == ' ' || c == '.')
                  {
                        if(word_in)
                        {
                              wordnum++;
                              word_in = 0;
                        }
                  }
                  else
                        word_in = 1;
            }
            if(prevletter == '¥0')
                  break;
            printf("ワード数：%d¥n", wordnum);
      }
}
```

処理手順を単純にするため、わざと無限ループにします。（while(1)）

[Enter] キーのときは、内側のループを抜けます。(単語が終わっていなければ単語数に加算します)

文字を判定して、単語の次の区切り文字なら、単語数をカウントアップします。

何も入力がなければ外側のループを抜けます。

1行分の処理（入力した文字を1つずつ処理します）

入力繰り返し

※太字はキーボードから入力した文字

実行結果

```
文字列を入力してください：I love cat. ↵
ワード数：3
文字列を入力してください：I love dog, too! ↵
ワード数：4
文字列を入力してください：▌
```

■ASCIIコード表を表示する

32〜127番のASCIIコード（16進、10進、キャラクタ）を表示します（0〜31番は画面に表示できない文字なので、表示しません）。

ソースコード

```
#include <stdio.h>

main()
{
    int x, y;           /* ループカウンタ */
    char c;             /* キャラクタ番号 */

    for(x = 2; x < 8; x++)
        printf("16: 10:c | ");
    printf("¥n");
    for(x = 2; x < 8; x++)
        printf("----------+-");
    printf("¥n");

    for(y = 0; y < 16; y++)
    {
        for(x = 2; x < 8; x++)
        {
            c = x * 16 + y;
            printf("%2X:%3d:%c | ", c, c, c);
        }
        printf("¥n");
    }
}
```

最上行の表示
16は16進、10は10進、cは文字の略

2行目

1行分の表示

実行結果

```
16: 10:c | 16: 10:c | 16: 10:c | 16: 10:c | 16: 10:c | 16: 10:c |
---------+----------+----------+----------+----------+-----------+-
20: 32:  | 30: 48:0 | 40: 64:@ | 50: 80:P | 60: 96:` | 70:112:p |
21: 33:! | 31: 49:1 | 41: 65:A | 51: 81:Q | 61: 97:a | 71:113:q |
22: 34:" | 32: 50:2 | 42: 66:B | 52: 82:R | 62: 98:b | 72:114:r |
23: 35:# | 33: 51:3 | 43: 67:C | 53: 83:S | 63: 99:c | 73:115:s |
24: 36:$ | 34: 52:4 | 44: 68:D | 54: 84:T | 64:100:d | 74:116:t |
25: 37:% | 35: 53:5 | 45: 69:E | 55: 85:U | 65:101:e | 75:117:u |
26: 38:& | 36: 54:6 | 46: 70:F | 56: 86:V | 66:102:f | 76:118:v |
27: 39:' | 37: 55:7 | 47: 71:G | 57: 87:W | 67:103:g | 77:119:w |
28: 40:( | 38: 56:8 | 48: 72:H | 58: 88:X | 68:104:h | 78:120:x |
29: 41:) | 39: 57:9 | 49: 73:I | 59: 89:Y | 69:105:i | 79:121:y |
2A: 42:* | 3A: 58:: | 4A: 74:J | 5A: 90:Z | 6A:106:j | 7A:122:z |
2B: 43:+ | 3B: 59:; | 4B: 75:K | 5B: 91:[ | 6B:107:k | 7B:123:{ |
2C: 44:, | 3C: 60:< | 4C: 76:L | 5C: 92:¥ | 6C:108:l | 7C:124:| |
2D: 45:- | 3D: 61:= | 4D: 77:M | 5D: 93:] | 6D:109:m | 7D:125:} |
2E: 46:. | 3E: 62:> | 4E: 78:N | 5E: 94:^ | 6E:110:n | 7E:126:~ |
2F: 47:/ | 3F: 63:? | 4F: 79:O | 5F: 95:_ | 6F:111:o | 7F:127:  |
```

COLUMN

コラム

～goto文～

breakやcontinueはちょっと反則してループを抜けたり、回を飛ばしたりする制御文でしたが、同じようなものに**goto**文というものがあります。gotoはその名前のとおり、指定した場所にジャンプする制御文で、これを使えば、複数のループを越えて、まったく別の場所に飛ぶことも可能になります。

goto文は次のようにして使います。

```
goto ラベル;
   ⋮
   ⋮
ラベル:文;
```

飛びたい文の先頭に「ラベル」をつけ、goto～でどこのラベルに飛びたいかを指定する。

このように紹介すると、なんだかとても便利なものに見えますが、実はgoto文には「プログラムが読みにくくなる」という大きな短所があるのです。

通常、プログラムというものは上から下へ流れてゆくものです。breakやcontinueはこれら制御文の含まれるブロックの中の流れが変わるだけですが、それがブロックやループを越えたところにまで影響してくるとなると、どうでしょう？　プログラムはたちまち無秩序でわかりにくいものになってしまいます。プログラムの混乱を避けるため、goto文の使用は極力避けるべきです。

2重ループを一気に抜け出したいときや、switch文を含むループから抜け出したいときなど、goto文は有用ですが、たいてい使わなくても同じことを実現できます。原則として「goto文は使用しない」と覚えておいてよいでしょう。

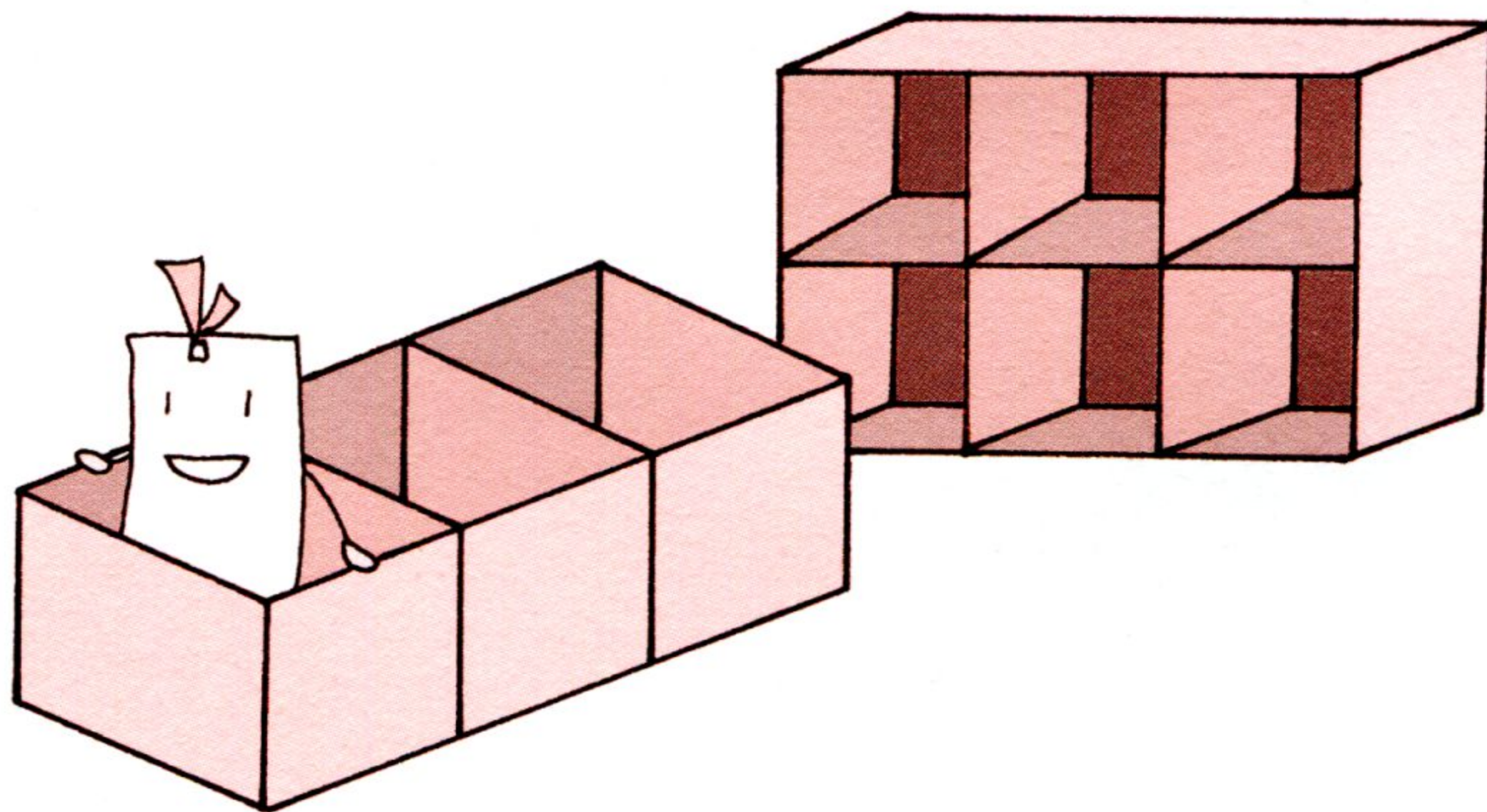
4 配列とポインタ

第4章

第4章は ここがkey!

Topics プログラムをより簡潔なものに

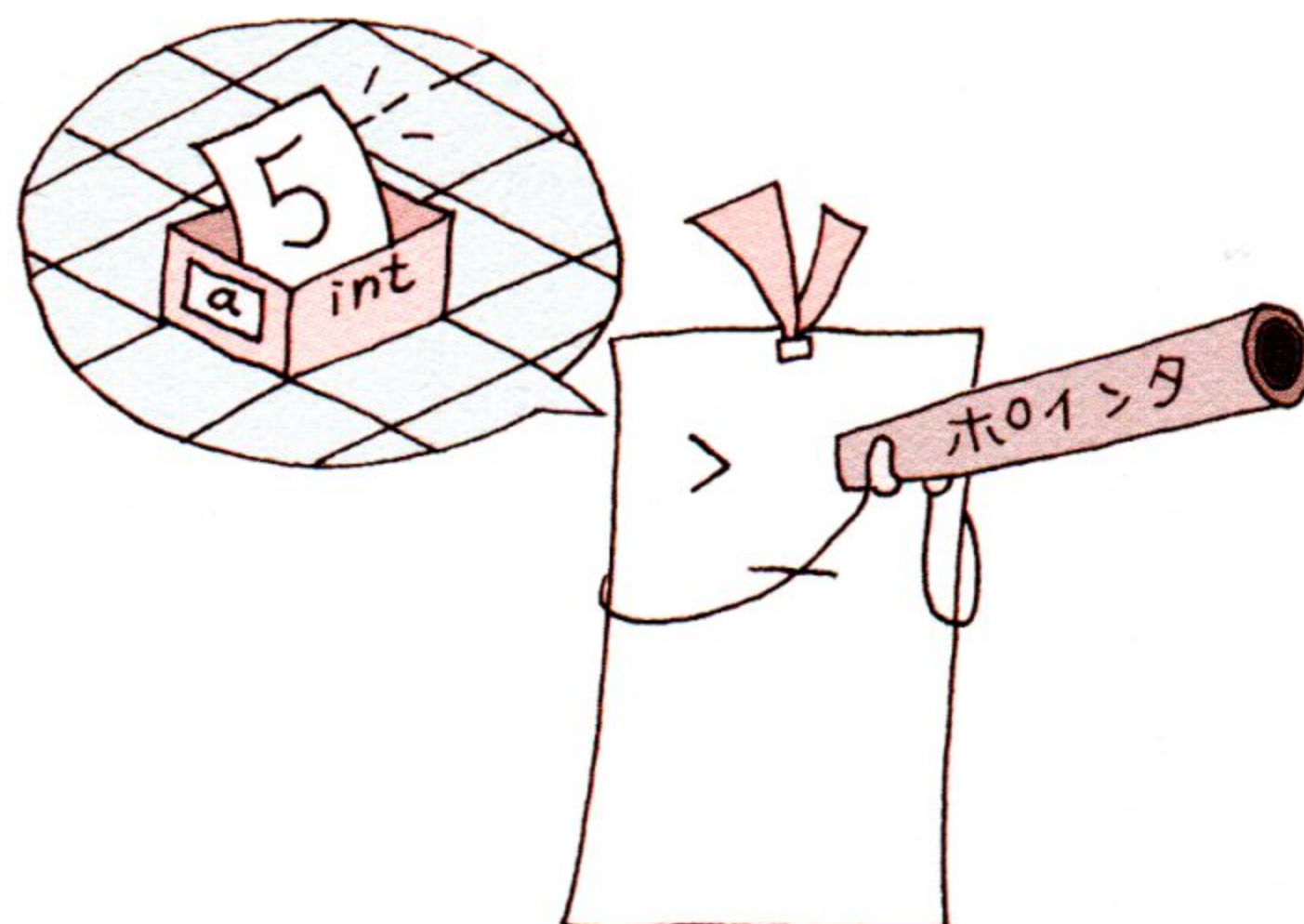
この章では**配列**と**ポインタ**について学んでいきます。配列は複数の変数を集めて一列に並べたものです。たとえば、`int a[4];`と宣言するとint型変数4個分を表します。この中の1つ分の箱（要素）を使うときは`a[0]`、`a[1]`、`a[2]`、`a[3]`といった0からはじまるインデックス番号を指定します。`a[i]`のように変数をインデックス番号に使えるので、a1、a2、…と変数を増やしていくよりも、宣言や処理を簡単にできます。

また、`int a[4]`という例では、変数を一列に並べた1次元のイメージでしたが、縦横に並べるイメージの2次元配列や、3次元配列、4次元配列などといったものもできます。配列の考え方は、大量のデータを管理する必要があるときは、なくてはならないものといえるでしょう。

ところで、第1章では、「文字列は配列である」と紹介しましたが、この章では配列と文字列の関係をもっと詳しく見ていきます。C言語には、文字列をコピーしたり、文字数を調べたりできる便利な関数がそろっていますので、それらについても紹介しておきましょう。

ポインタと配列は切っても切れない関係？

ポインタとは、「データのありかを保管しておくための変数」です。それだけ聞くと、なかなかピンときませんが、こんなたとえ話はどうでしょう。あなたは、久しぶりにお気に入りのレストランに行ってみることにしました。しかし、行ってみたらそこにお店はなく、代わりに移転先が書いてある看板があるだけでした。看板の住所に行ってみると、ちゃんとレストランがあり、大好きな料理を食べることができました。このときの看板にあたるのがポインタです。レストランは変数で、料理はその中のデータとすると、ポインタとそれが指し示す変数、そしてその中に入っているデータの関係が少しはイメージできたでしょうか？

実際には、変数やポインタはメモリ上に存在していますので、少しコンピュータの内部のこともお話しする必要があります。メモリには**アドレス**と呼ばれる数字がついていて、メモリ上に存在するデータは、その場所をアドレスで表すことができるのです。それならポインタなんていわずに、アドレスといった方が簡単じゃないの？と思うかもしれませんが、いろいろな種類のコンピュータがある中で、内部のしくみにあまり依存してしまうのはよくありません。また、実はポインタと配列には密接な関係があります。その考え方についてもお話していきます。

「以前C言語を勉強してみたけどポインタでつまずいた」という人も「なんだかややこしそうだな」と思っている人も、じっくりと配列とポインタについて学んでいきましょう。

配 列

同じ型のデータであれば配列としてまとめて扱うことができます。

配列の考え方

配列は複数の同じ型の変数を1つにまとめたものです。大量のデータを扱うときや複数のデータを次々と自動的に読み出したいときは配列を使うと便利です。
配列の宣言は次のように行います。

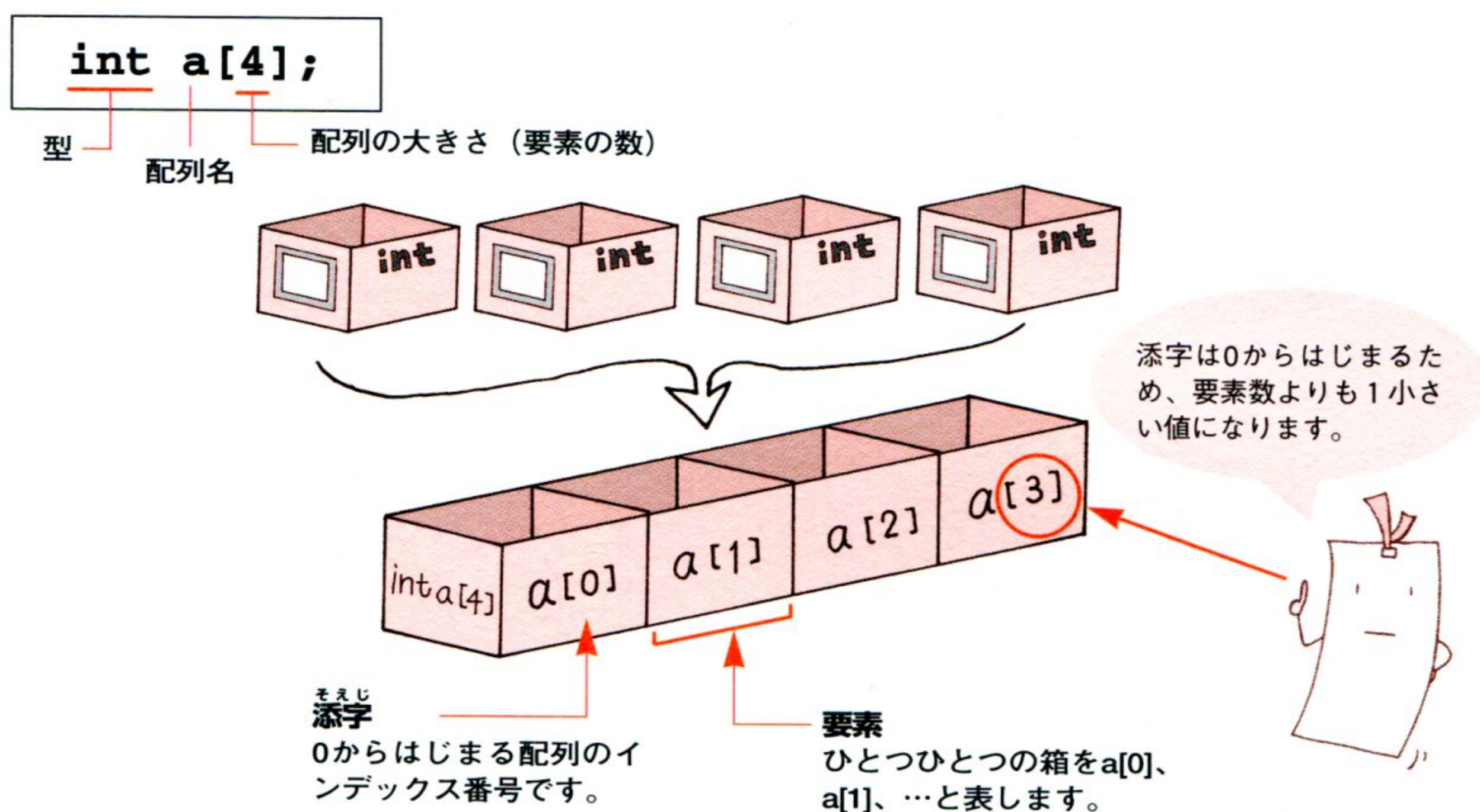
```
int a[4];
```

型　配列名　配列の大きさ（要素の数）

int a[4]　a[0]　a[1]　a[2]　a[3]

添字は0からはじまるため、要素数よりも1小さい値になります。

添字（そえじ）
0からはじまる配列のインデックス番号です。

要素
ひとつひとつの箱をa[0]、a[1]、…と表します。

配列を初期化するときは、{ }を使って値を列挙します。

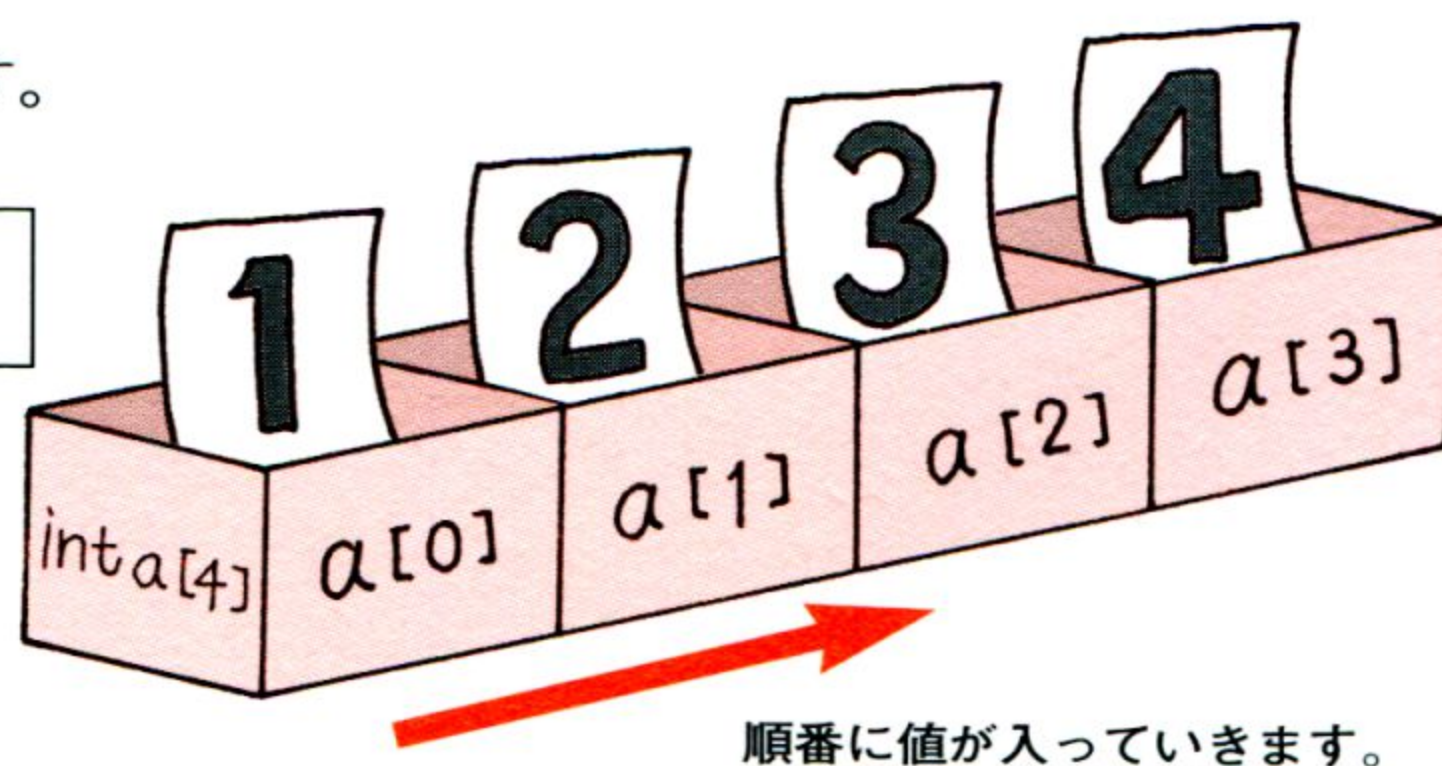
```
int a[4] = {1, 2, 3, 4};
```

[ ] 内の要素数は省略することができます。

```
int a[] = {1, 2, 3, 4};
```

{ }内にデータがいくつあるかで要素数を自動的に決定します。

1　2　3　4
int a[4]　a[0]　a[1]　a[2]　a[3]

順番に値が入っていきます。

配列要素の参照と代入

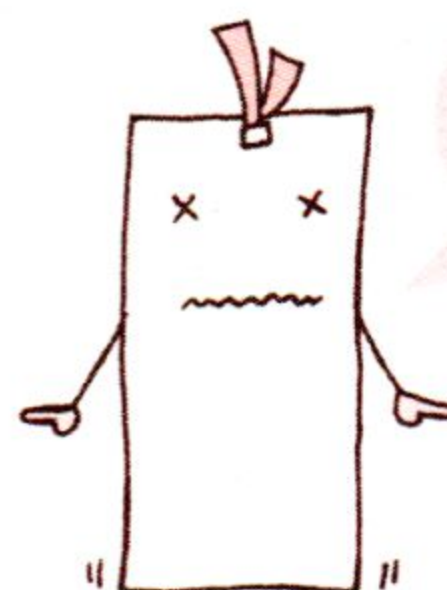
配列の要素ひとつひとつは、普通の変数のように参照と代入ができます。

```
int a[4];
int n = 1;

a[0] = 1;
a[1] = 2;
a[2] = 3;
a[3] = 4;
printf("%d¥n", a[n]);
```

a[0]〜a[3]の値を代入

a[1]の値を表示

添字の数に「0」〜「要素数－1」以外の値を指定すると、
実行時にエラーになるので要注意！

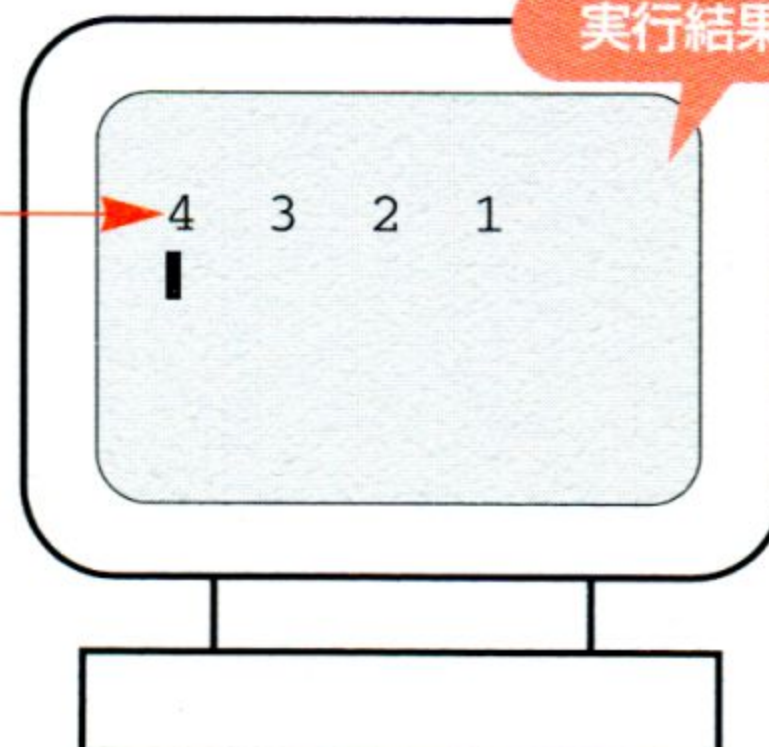
```
int a[4] = {1, 2, 3, 4};
printf("%d", a[4]);
```

a[4]は配列の範囲外なので、プログラムが途中で止まったり、予期しない動きをしてしまいます。

例

```
#include <stdio.h>

main()
{
    int i;
    int a[] = {1, 2, 3, 4};

    for(i = 3; i >= 0; i--)
        printf("%d  ", a[i]);
    printf("¥n");
}
```

添字に変数を使って表示を自動化しています。

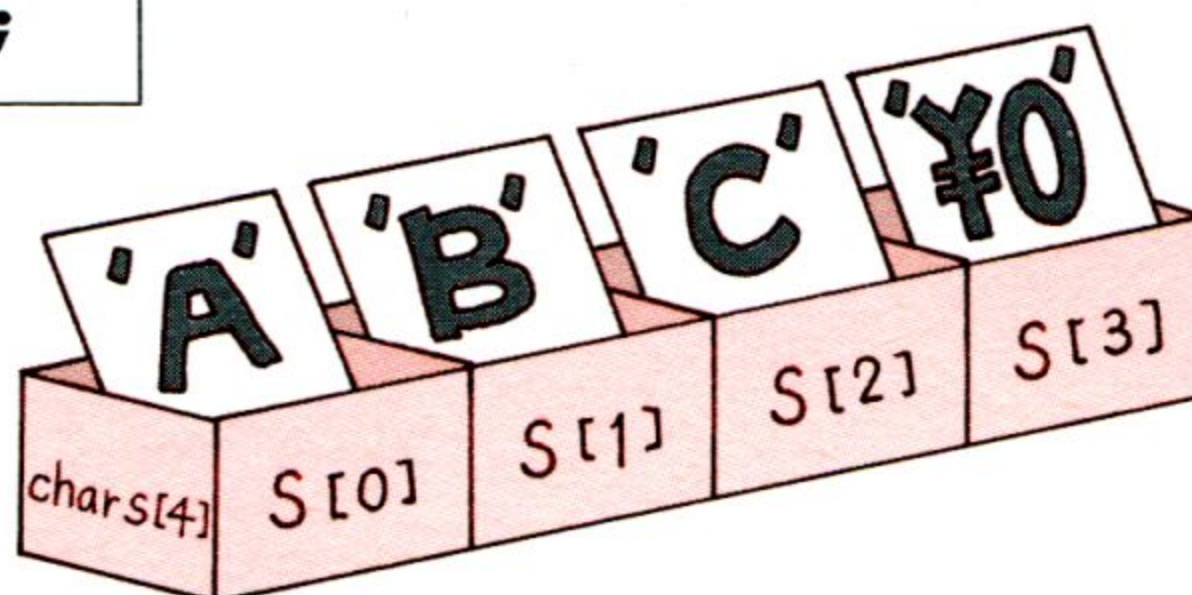
実行結果

```
4   3   2   1
```

配列と文字列

配列は同じ型のデータをまとめて格納するものですが、文字型を集めたものが文字列です。

配列と文字列の関係

文字列は複数の文字の集まりで、これを格納するには配列（**文字列配列**）を使うことを第1章で学びました。文字列では配列の要素の1つに1文字が入っています。

```
char s[] = "ABC";
```

配列の初期化に習って、次のように書いても同じです。

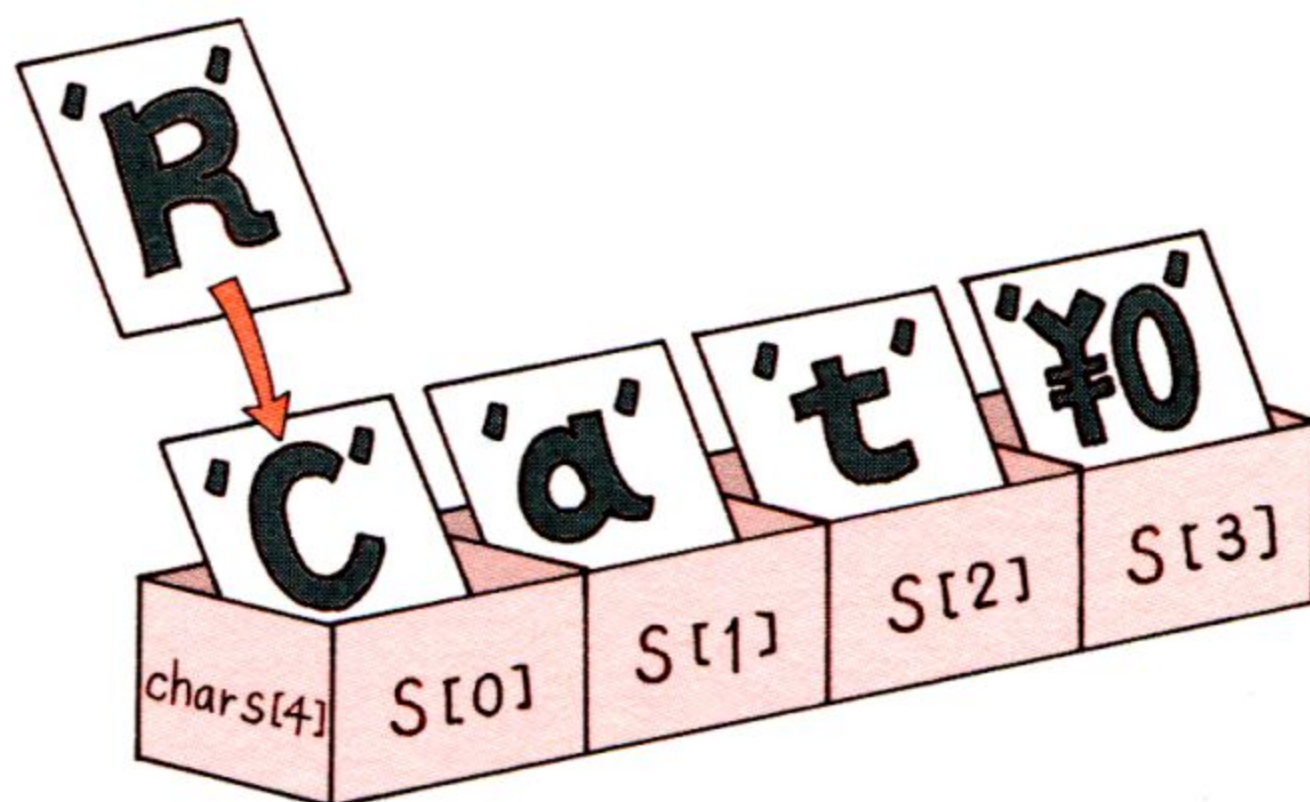
```
char s[4] = {'A', 'B', 'C', '¥0'};
```

NULL文字が必要です。

文字単位の操作

配列の考え方を用いると、次のように指定した1文字のみ入れ替えることもできます。

```
char s[4] = "Cat";
s[0] = 'R';
```

0番の箱に'R'を代入します。

すでにデータの入っているところに上書きできます。

例

```
#include <stdio.h>

main()
{
    int i = 0;
    char a[] = "NET";
    char b[4];

    while(a[i] != '¥0')
    {
        b[i] = a[2-i];
        i++;
    }
    b[3] = '¥0';
    printf("%sは逆から読むと%s¥n", a, b);
}
```

NULL文字'¥0'がきたら繰り返しを終了します。

文字列の最後は必ず'¥0'です。

実行結果

```
NETは逆から読むとTEN
```

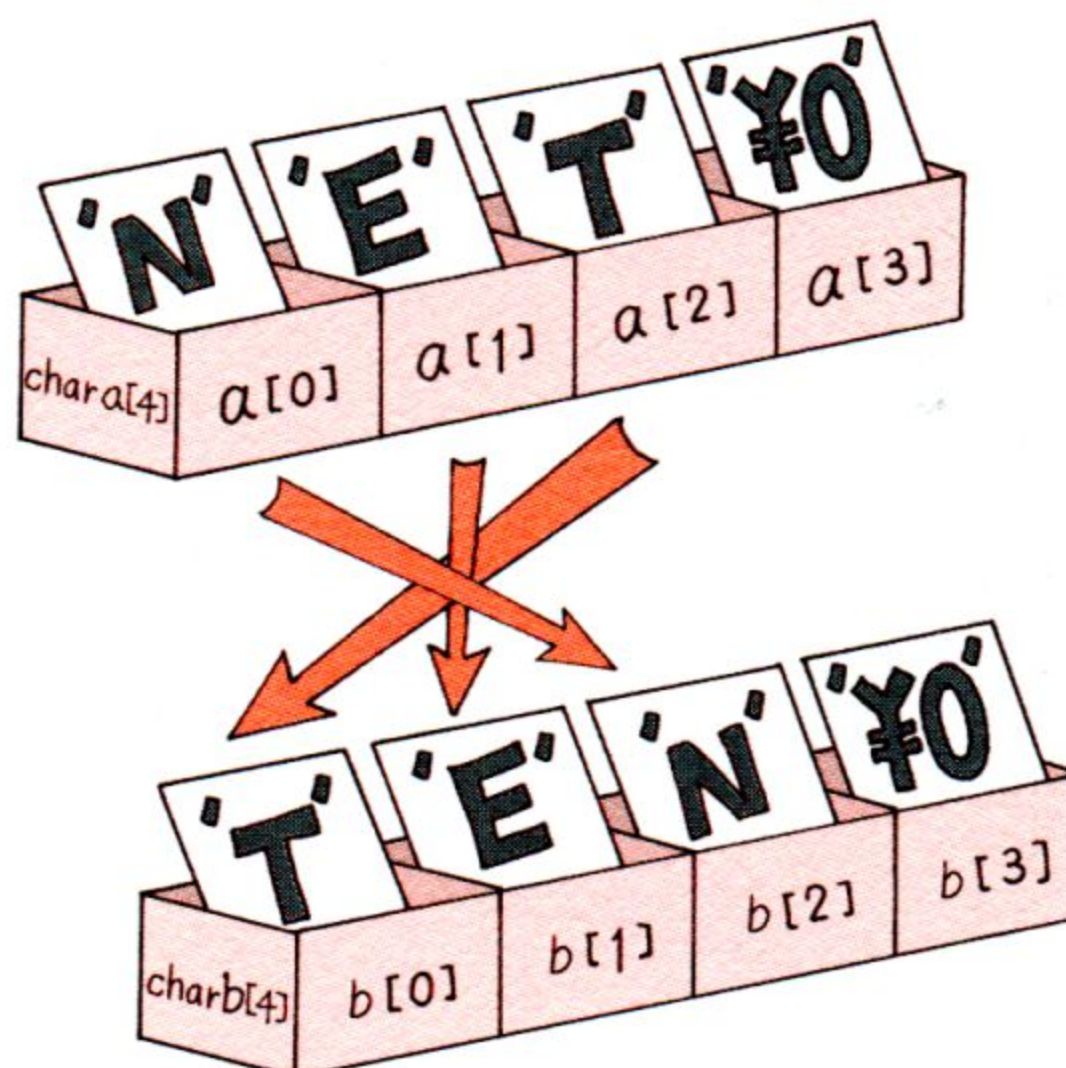
i = 0　b[0] = a[2];

i = 1　b[1] = a[1];

i = 2　b[2] = a[0];

b[3] = '¥0';

'N' 'E' 'T' '¥0'
char a[4]　a[0]　a[1]　a[2]　a[3]

'T' 'E' 'N' '¥0'
char b[4]　b[0]　b[1]　b[2]　b[3]

文字列自由自在

C言語に標準でついている、文字列を活用するための便利な関数について見ていきます。

文字列関数

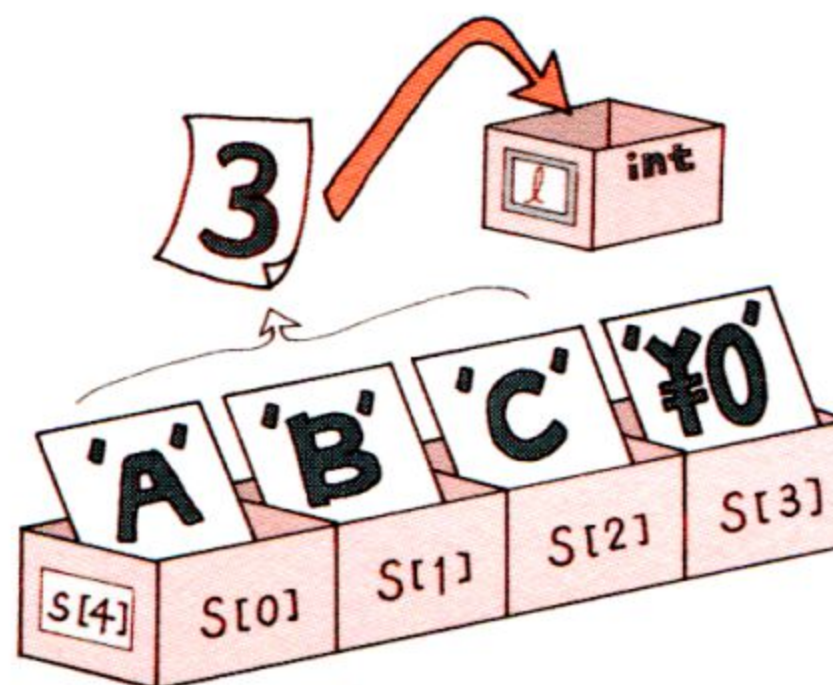
C言語には標準で、文字列を操作する関数（**文字列関数**）が用意されています。文字列関数を使うにはプログラムの先頭に次の記述を追加する必要があります。

```
#include <string.h>
```

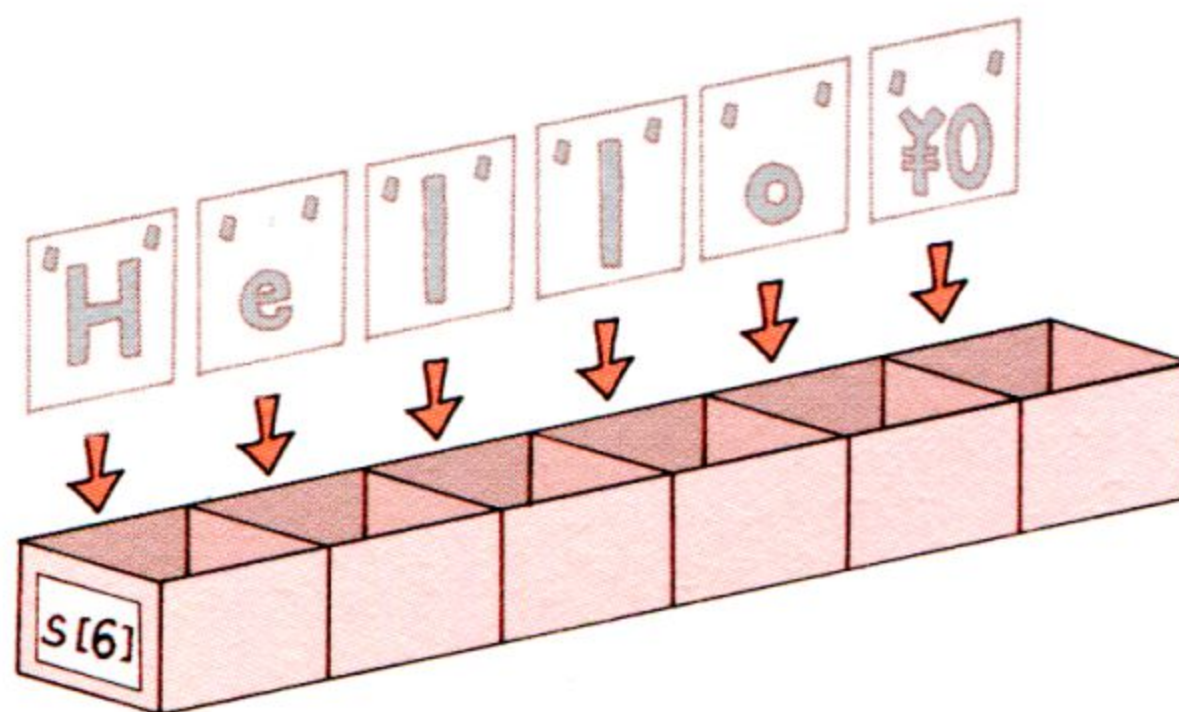
代表的な文字列関数には次のようなものがあります。

strlen()（ストリングレン）　文字列の長さを得る

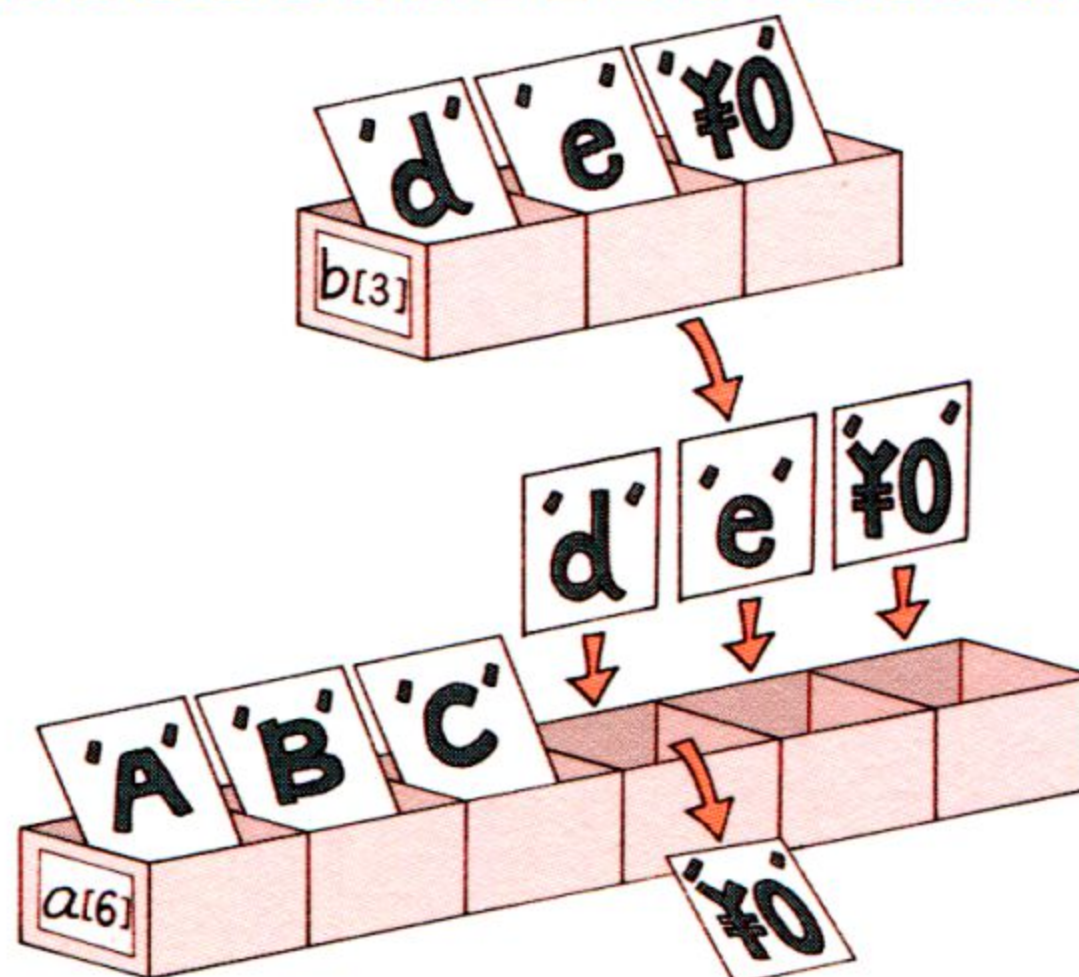
```
char s[] = "ABC";
int l;
l = strlen(s);
```

変数 l に文字列sの長さを代入します。

strcpy()　文字列をコピーする

```
char s[6];
strcpy(s, "Hello");
```

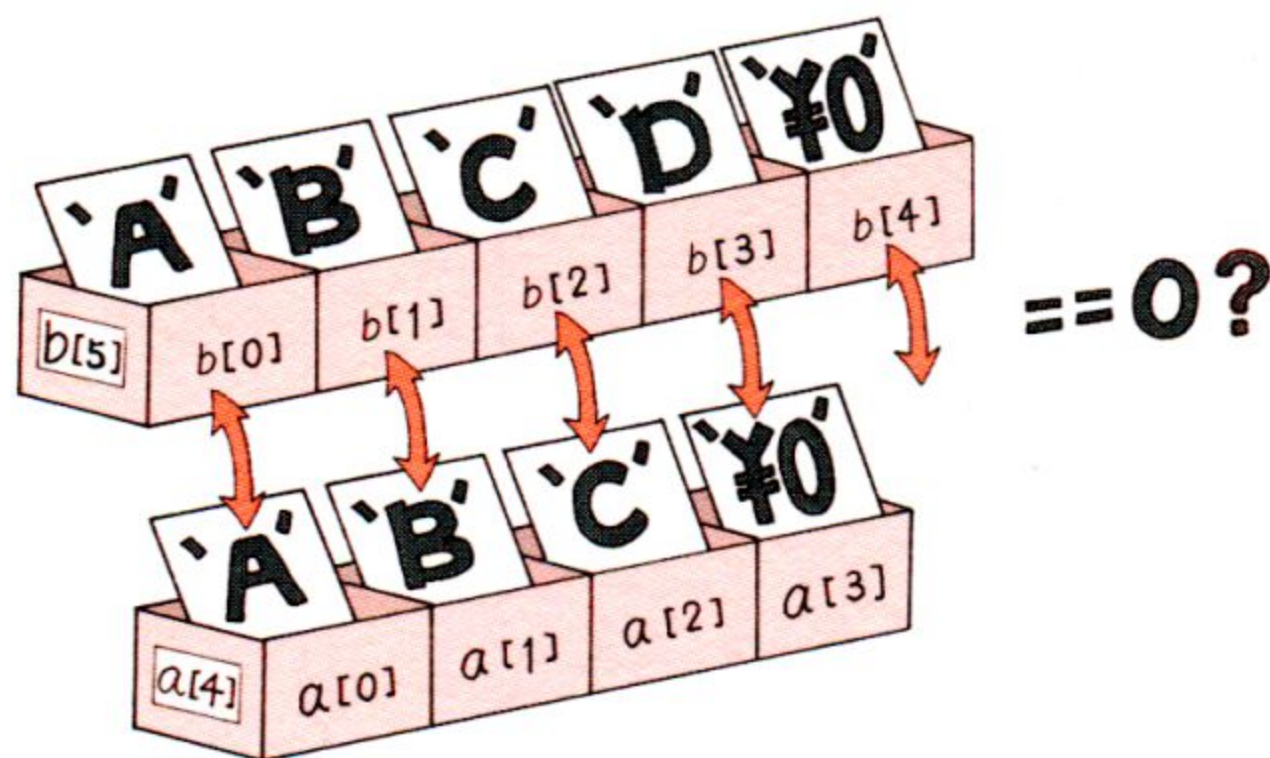
文字列sに文字列"Hello"をコピーします。

strcat()（ストリングキャット）　文字列を結合する

```
char a[6] = "ABC";
char b[] = "de";
strcat(a, b);
```

文字列aに文字列 b を結合します。

strcmp()（ストリングコンプ） 文字列を比較する

```
char a[] = "ABC";
char b[] = "ABCD";
int c = strcmp(a, b);
```

文字列aと文字列bを比較した結果をcに代入します(ただし文字の比較はASCIIコードに基づき、大文字と小文字を区別します)。
この例の場合、cは負の値になります。

b[0] b[1] b[2] b[3] b[4] b[5]
a[0] a[1] a[2] a[3] a[4]
==0?

代入されるcの値	意味
c == 0	aとbは等しい
c > 0	aの方がbよりも辞書的に後
c < 0	aの方がbよりも辞書的に前

C 文字列変換関連の関数

次のような変換系の関数もよく使われます。

sprintf()（エスプリントエフ） 数値などを文字列に変換する

```
char s[40];
sprintf(s, "%f", 143.5);
```

printf( )の要領で文字列に変換した結果をsに格納します。

atoi()（エイツゥーアイ） 文字列を数値に変換する

```
char s[] = "340";
int n = atoi(s);
```

sを10進の整数に変換した結果をnに代入します(#include <stdlib.h>が必要)。

例

```
#include <stdio.h>
#include <string.h>

main()
{
    char s1[] = "cat", s2[] = "dog";
    char s[20];
    sprintf(s, "I love %s and %s.", s1, s2);
    printf("「%s」の文字数は%d¥n", s, strlen(s));
}
```

実行結果

```
「I love cat and dog.」の文字数は19
```

多次元配列

表など、縦横の広がりのあるデータを一度に扱うには多次元配列が大変便利です。

多次元配列とは？

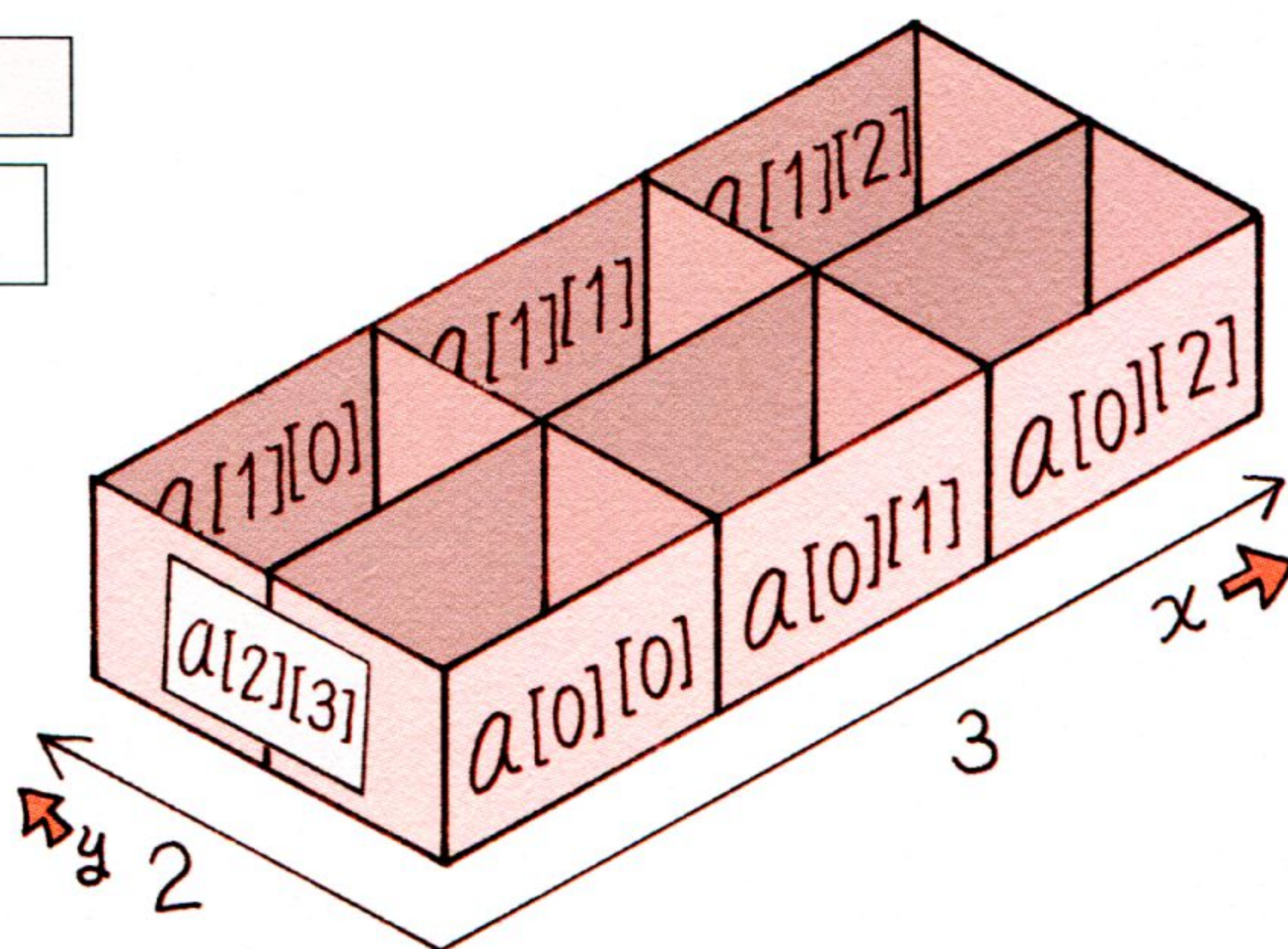
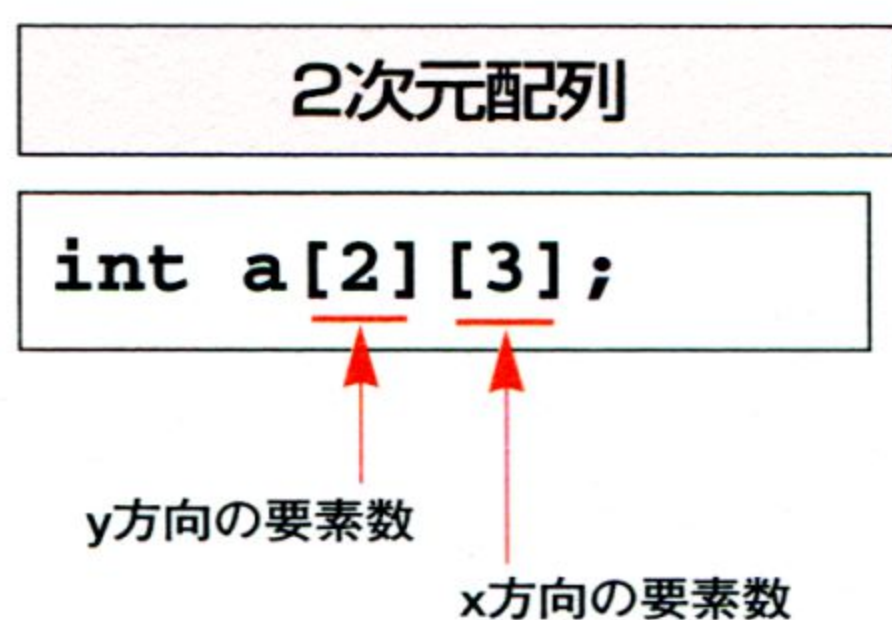
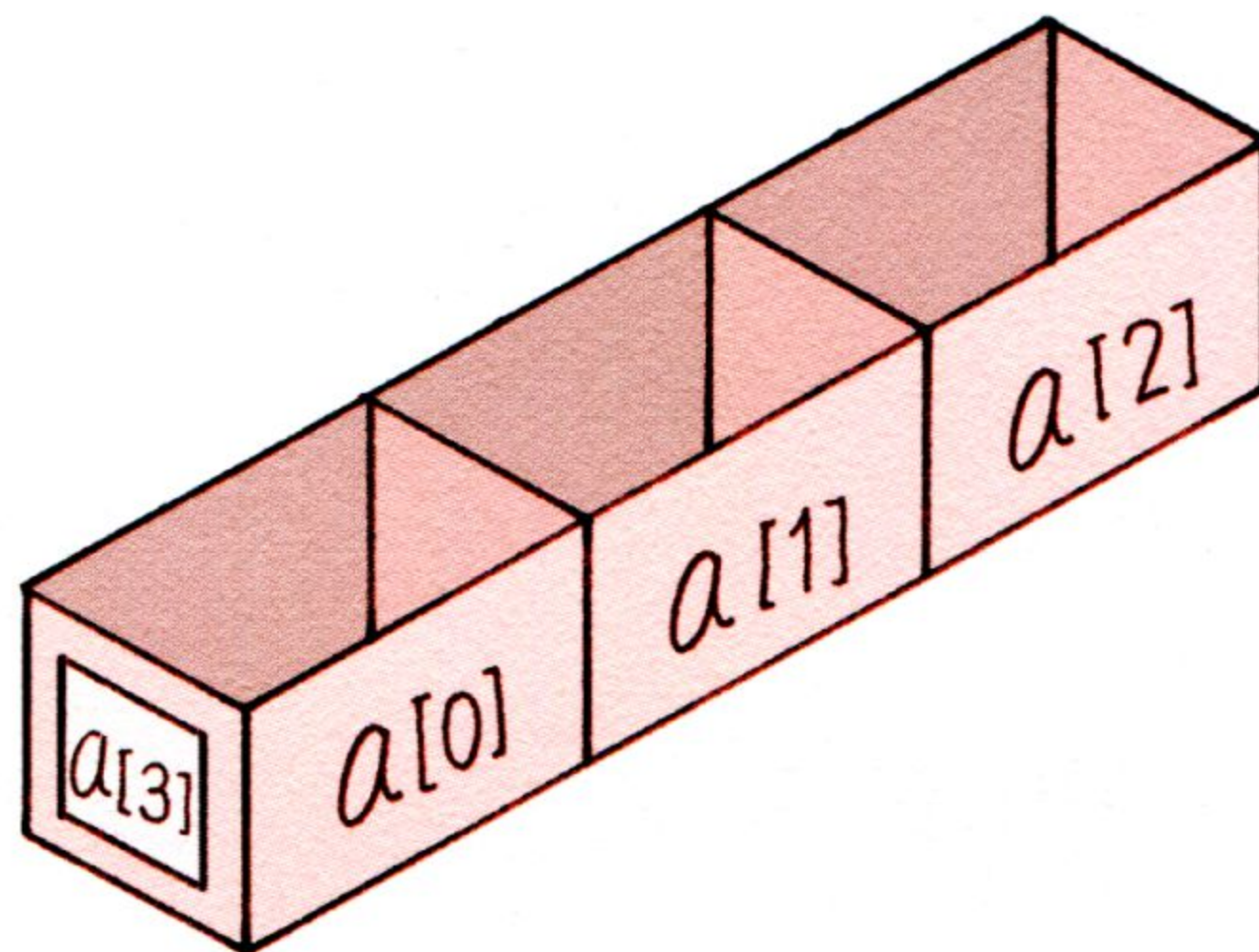
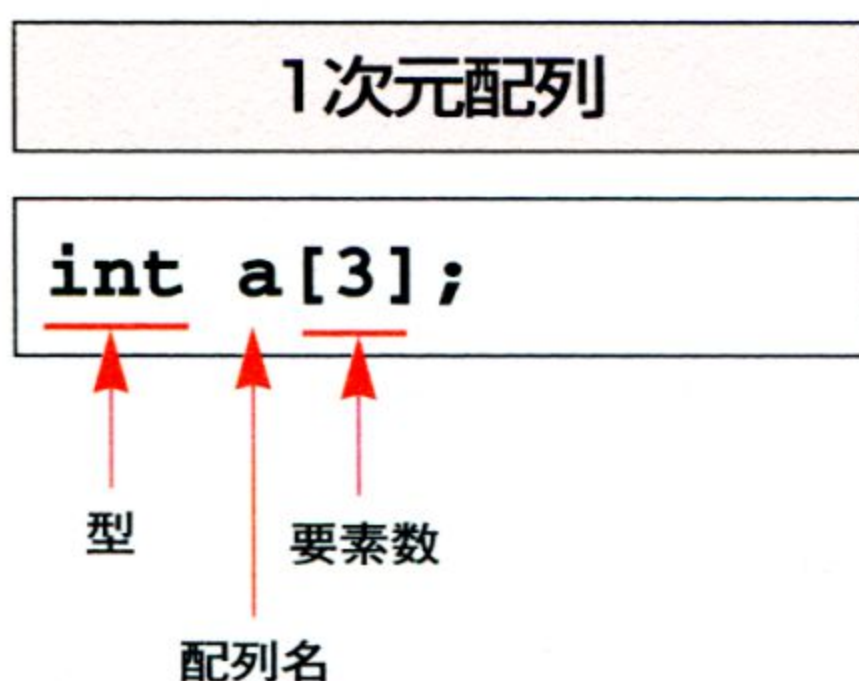
今までの配列は要素数に応じて横に伸びていく1次元のイメージでしたが、今度は2次元、3次元的なものを考えてみます。

1次元配列

```
int a[3];
```

型　配列名　要素数

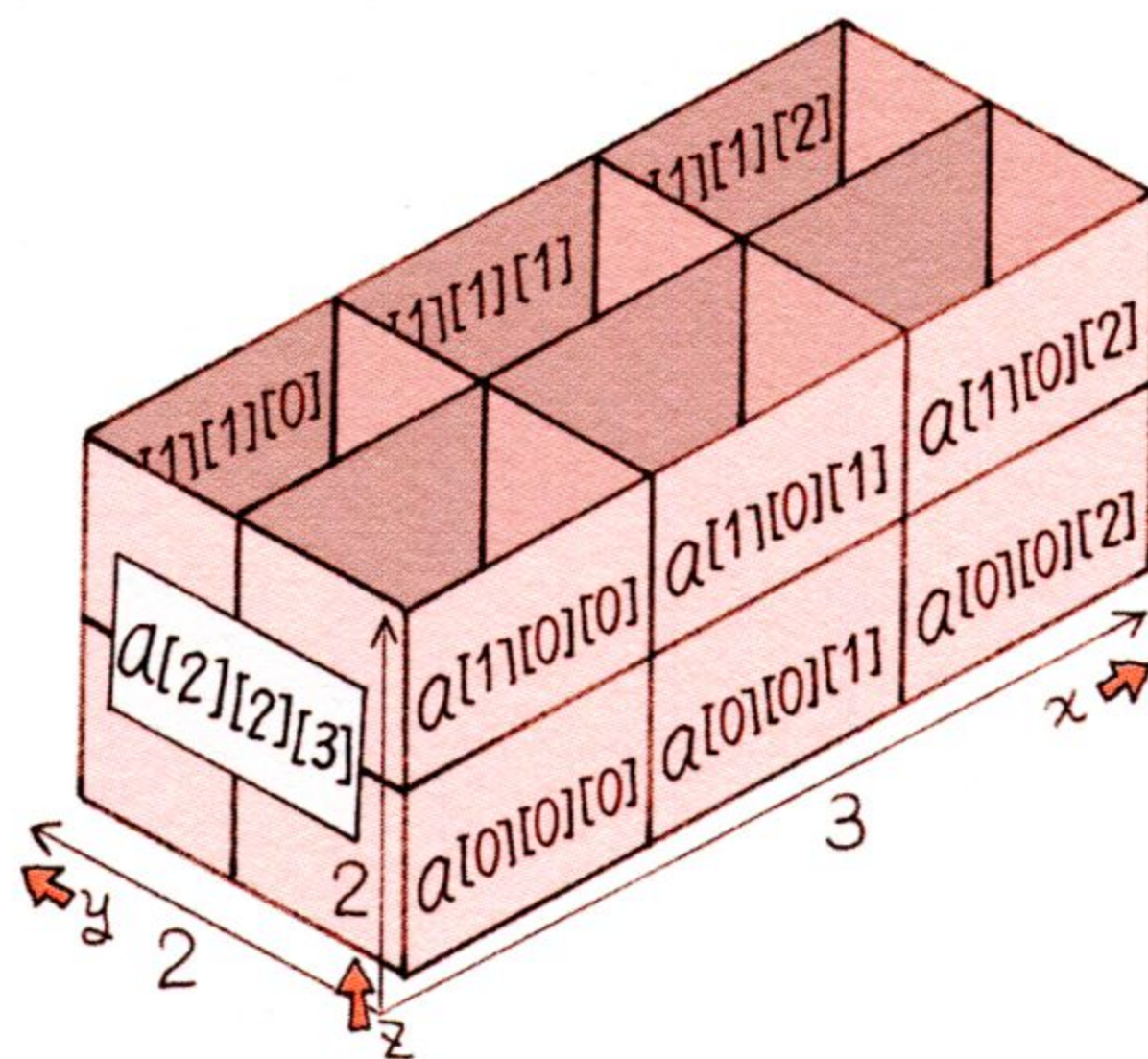
2次元配列

```
int a[2][3];
```

y方向の要素数　x方向の要素数

3次元配列

```
int a[2][2][3]
```

z方向の要素数
y方向の要素数
x方向の要素数

配列の次元は必要に応じて4次元、5次元と増やせますが、要素の数が増えればその分メモリを消費し、システムに負担をかけることになるので注意してください。

C 多次元配列への代入・初期化・参照

多次元配列への代入、初期化、参照は次のように行います。

```
int a[2][3] = {
    {10, 20, 30},
    {40, 50, 60}
};
a[0][2] = 0;
printf("%d¥n", a[1][0]);
```

初期化
{}やカンマの組み合わせに注意。

a[0][2]に0を代入。

a[1][0]を参照。

例

```
#include <stdio.h>

main()
{
    int x, y;
    int a[2][3] = {
        {10, 20, 30},
        {40, 50, 60}
    };
    for(y = 0; y < 2; y++){
        for(x = 0; x < 3; x++)
            printf("a[%d][%d] = %d ", y, x, a[y][x]);
        printf("¥n");
    }
}
```

実行結果

```
a[0][0]=10 a[0][1]=20 a[0][2]=30
a[1][0]=40 a[1][1]=50 a[1][2]=60
```

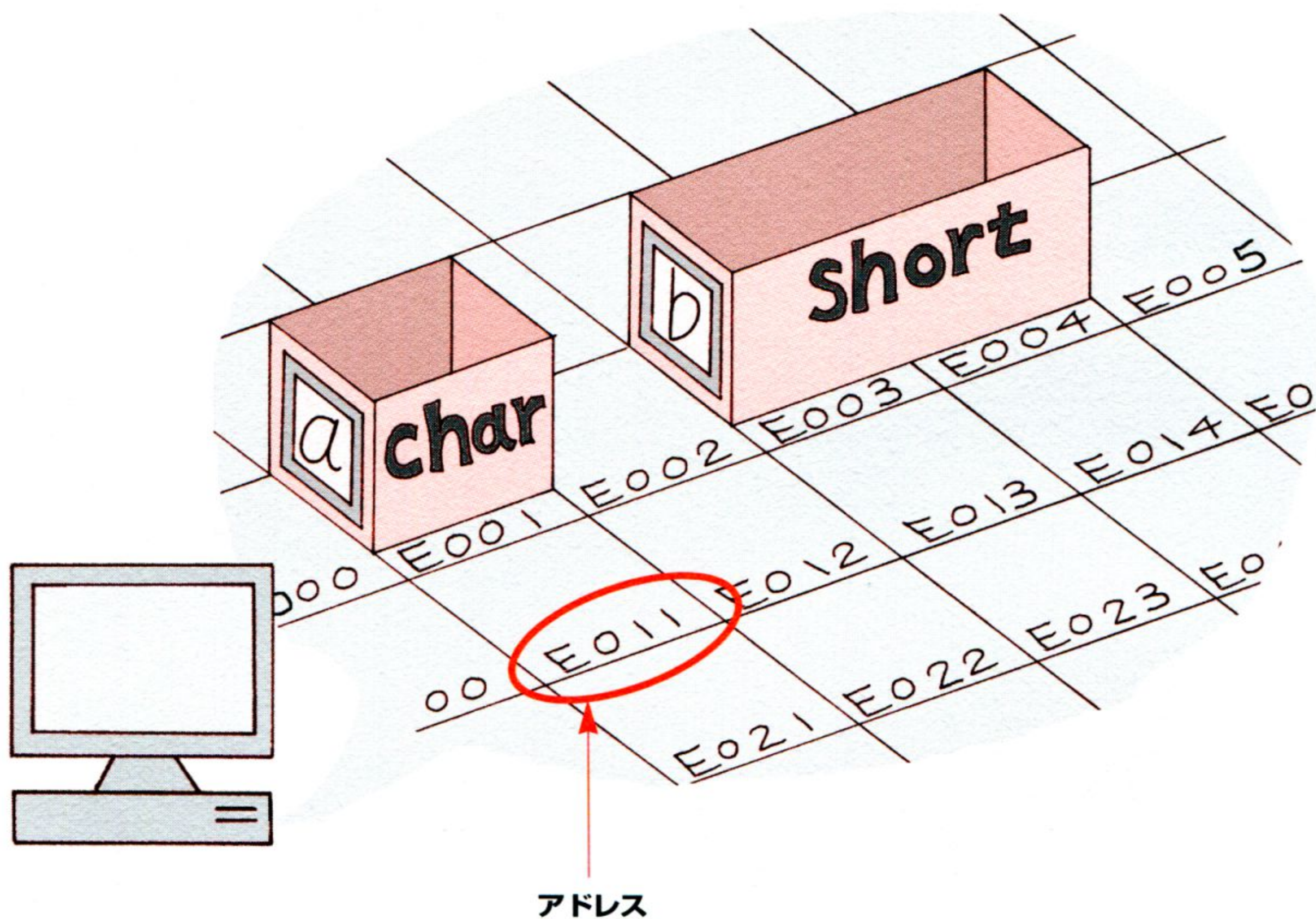
それぞれのデータがどの位置に格納されているかに注意しましょう。

アドレス

メモリには1バイトごとに番号が振ってあり、データがどこにあるかはこの番号で表すことができます。

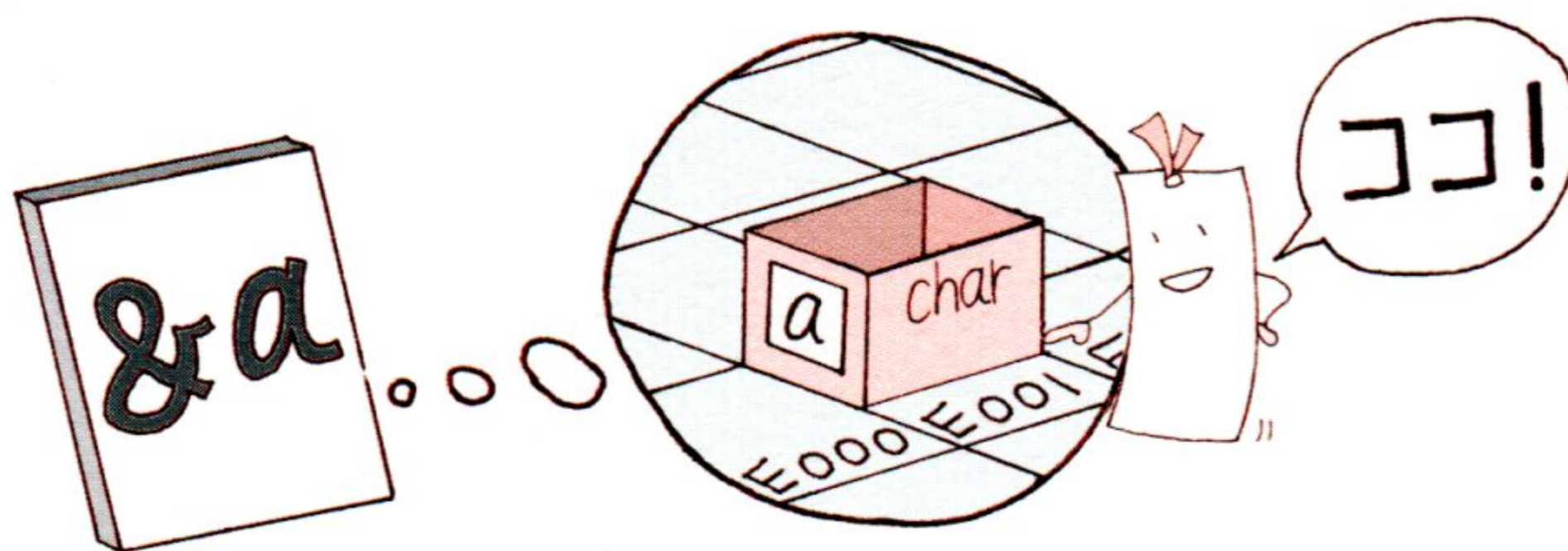
アドレスとは？

変数や配列は、実際にはコンピュータのメモリ上にあります。つまり、変数や配列の値はメモリ上に記録されているのです。メモリにはアドレスという連続した番号がついていて、どこに何が入っているかを管理できるようになっています。

※アドレスは実際にはもっと桁が大きい場合もありますが、ここでは説明のため、4桁の16進数で表しています。

アドレス

ひとマスで1バイトです。

アドレスの表し方

変数名の頭に&をつけるとその変数の位置（アドレス）を表します。

つまり左ページのようにデータが存在する場合は、

&a = 0xE001
&b = 0xE003

となります。

例

```
#include <stdio.h>

main()
{
    char a;
    short b;

    printf("aのアドレスは%x、bのアドレスは%x ¥n", &a, &b);
}
```

実行結果

```
aのアドレスはxxxx、bのアドレスはxxxx
```

アドレスが入ります。
（実行環境により表示される値は異なります）

ポインタ

データのある場所を記憶するポインタ変数を紹介します。

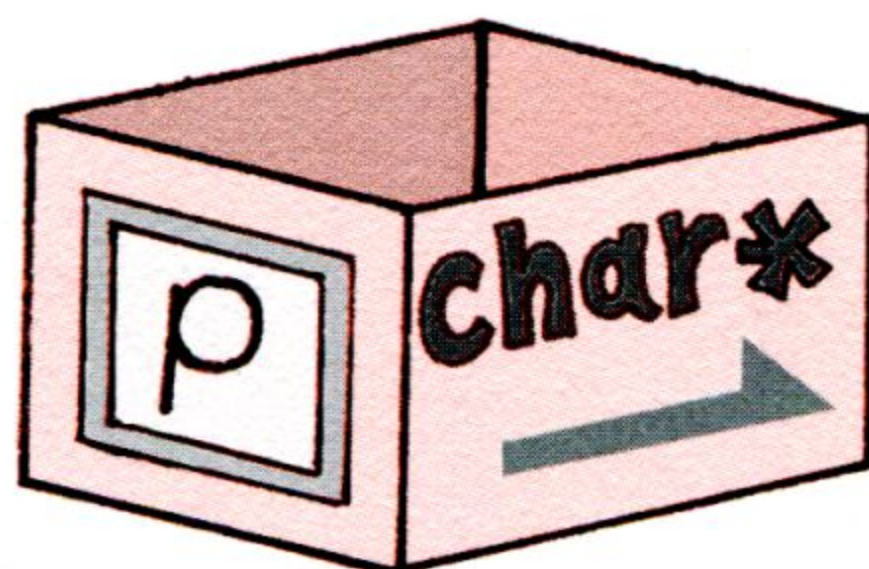
ポインタとは？

変数などが格納されている位置（アドレス）を値とする変数をポインタといいます。ポインタにも型の区別があり、たとえばchar型のポインタ変数pを宣言するには次のようにします。

ポインタpの宣言

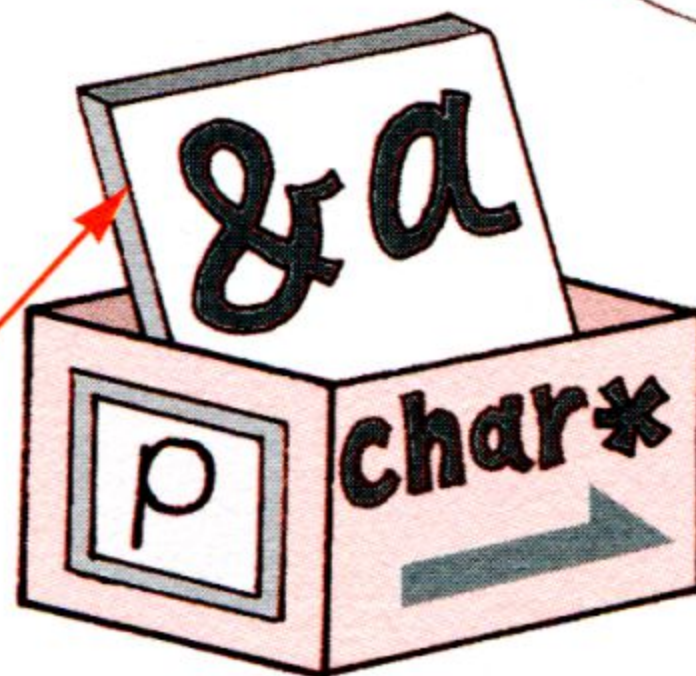
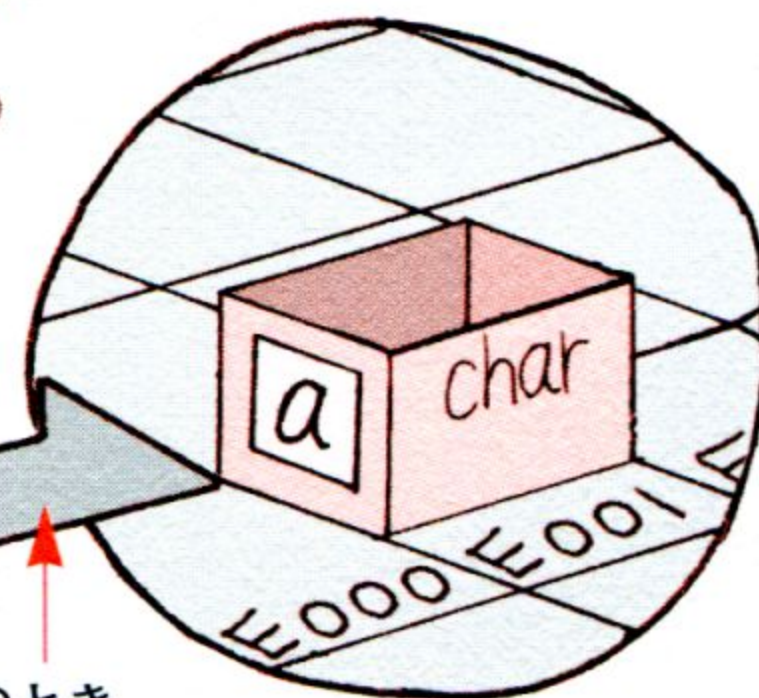
```
char *p;
```

または

```
char* p;
```


ポインタへのアドレスの代入は次のように行います。

ポインタpに変数aのアドレスを代入

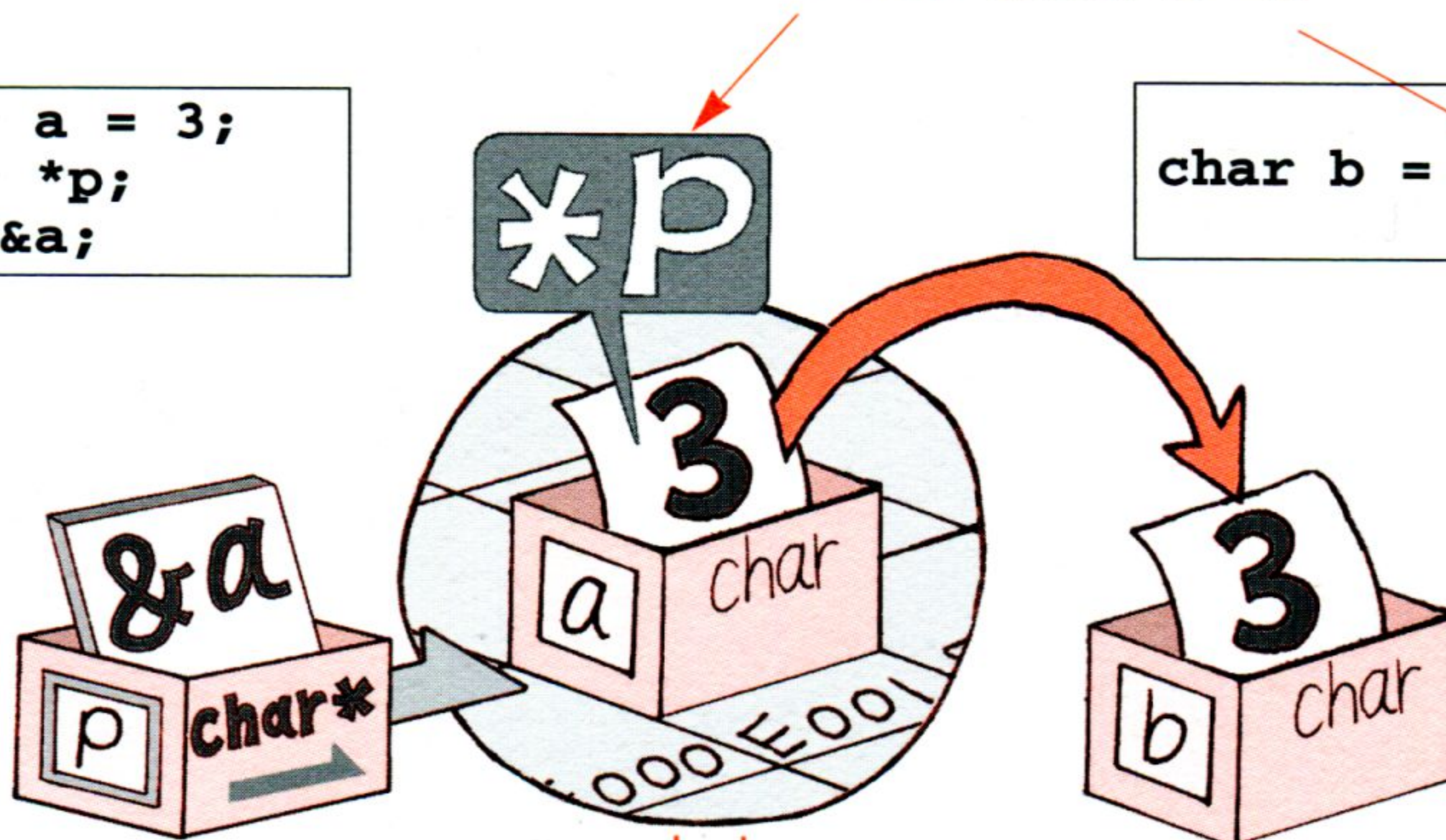
```
char a;
char *p;
p = &a;
```

C ポインタが指す値の参照

ポインタ名の前に* をつけると、そのポインタが指す先のデータを参照します。

ポインタpが指す変数aの値を参照

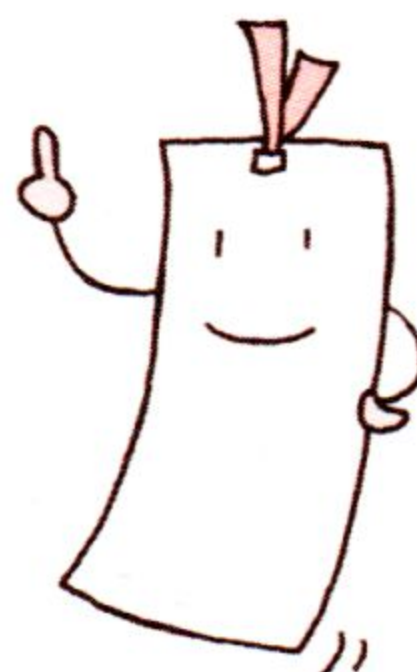
```
char a = 3;
char *p;
p = &a;
```

```
char b =  *p;
```

ポインタpの指す変数（a）の値

ポインタpに変数aのアドレスを代入。

ポインタpを使って、変数aの値を変数bに代入。

b = *p の「*」と、
char *p の「*」は
意味が異なります。

例

```
#include <stdio.h>

main()
{
    char x = 4, y;
    char *p = &x;
    y = *p;
    printf("変数xの値は%dです¥n",y);
}
```

実行結果

```
変数xの値は4です
```

NULLポインタ

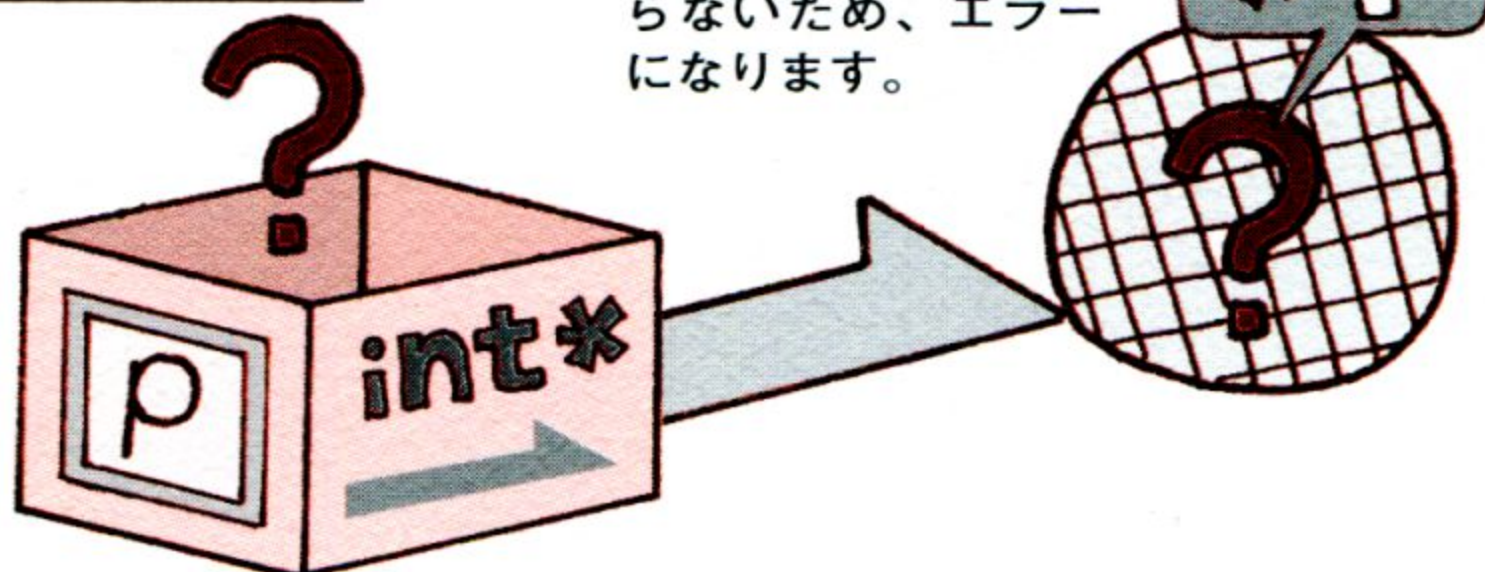
ポインタの使用上の注意点や、何も指さないNULLポインタを紹介します。

間違った参照

ポインタを利用するときは必ずその値が指し示すアドレスに、データが存在している必要があります。ポインタを初期化しないで使うと、何もないところを指し示すことになり、実行時エラーの原因となります。

ポインタpが指す値を参照？

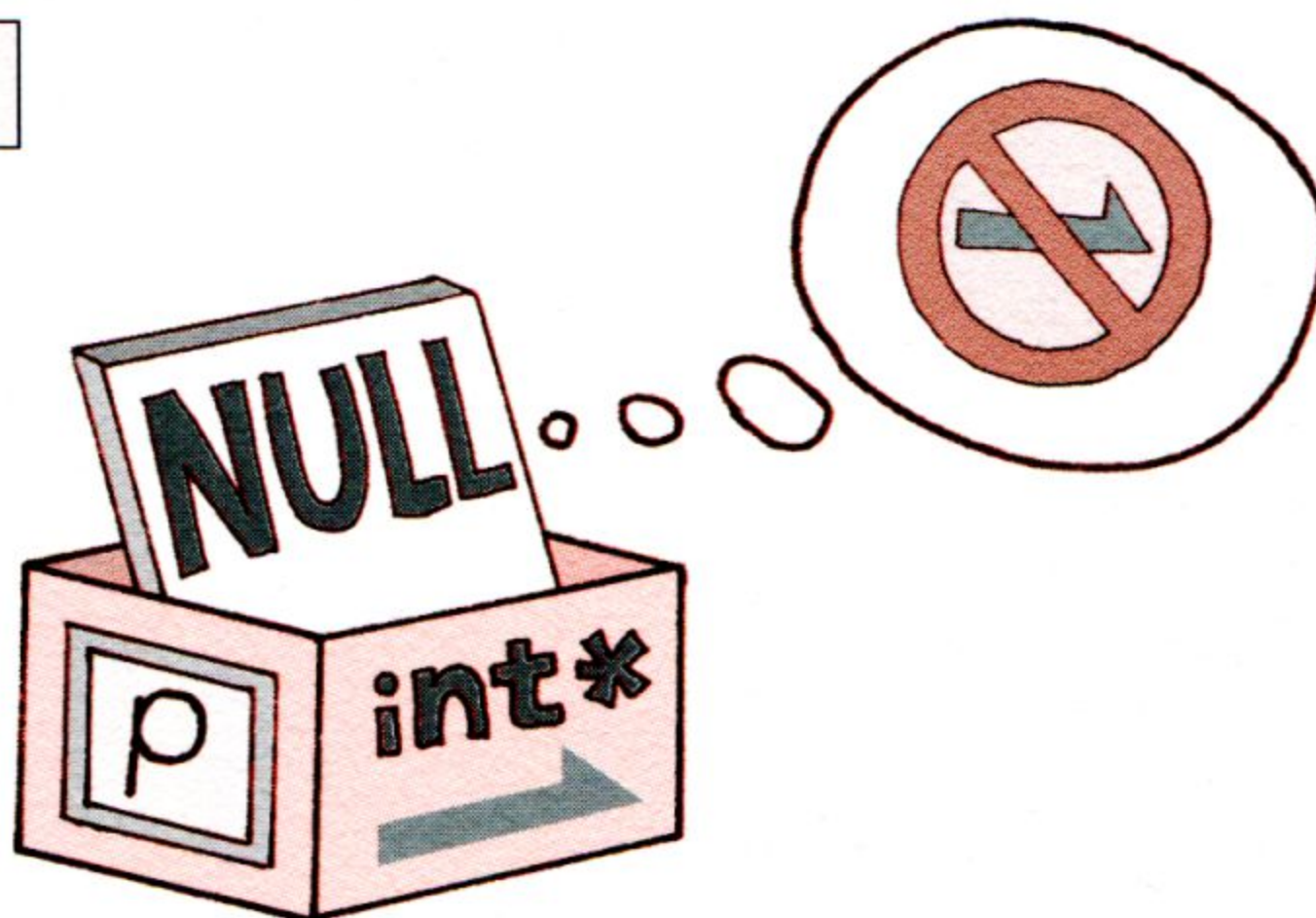
```
int a;
int *p;
a = *p;
```

ポインタpがどこを指しているのかわからないため、エラーになります。

ポインタは値を設定しないと使えません。

NULLポインタ

プログラムの中で、どこも指し示していないことをはっきりさせたいときは、**NULL（ヌル）ポインタ**を使います。NULLポインタはどの型のポインタにも格納できます。

ポインタpをNULLで初期化

```
int *p = NULL;
```

ポインタpが有効かどうかを調べるには、次のようにします。NULLはポインタ版の「0」のようなものですが、実は実際の値も0になっています。ですから、論理演算の手法でも書けます。

pは有効？	`if(p != NULL)`	または	`if(p)`
pは無効？	`if(p == NULL)`	または	`if(!p)`

falseの値も0でしたね。

例

```
#include <stdio.h>
#include <string.h>

main()
{
    char s[] = "I love cat.";
    char c = 'd';
    char *p = NULL;

    printf("文字列「%s」の中に文字「%c」", s, c);
    p = strchr(s, c);
    if(!p)
        printf("はありません。¥n");
    else
        printf("を発見しました。¥n");
}
```

実行結果

```
文字列「I love cat.」の中に文字「d」はありません。
```

strchr()関数

（ストリングチャー）

指定した文字が文字列内に存在するか検索します。

・存在する場合は、最初にその文字が現れた位置のポインタを返します。

・存在しない場合は、NULLを返します。

最初のl

'H' 'e' 'l' 'l' 'o' '¥0'

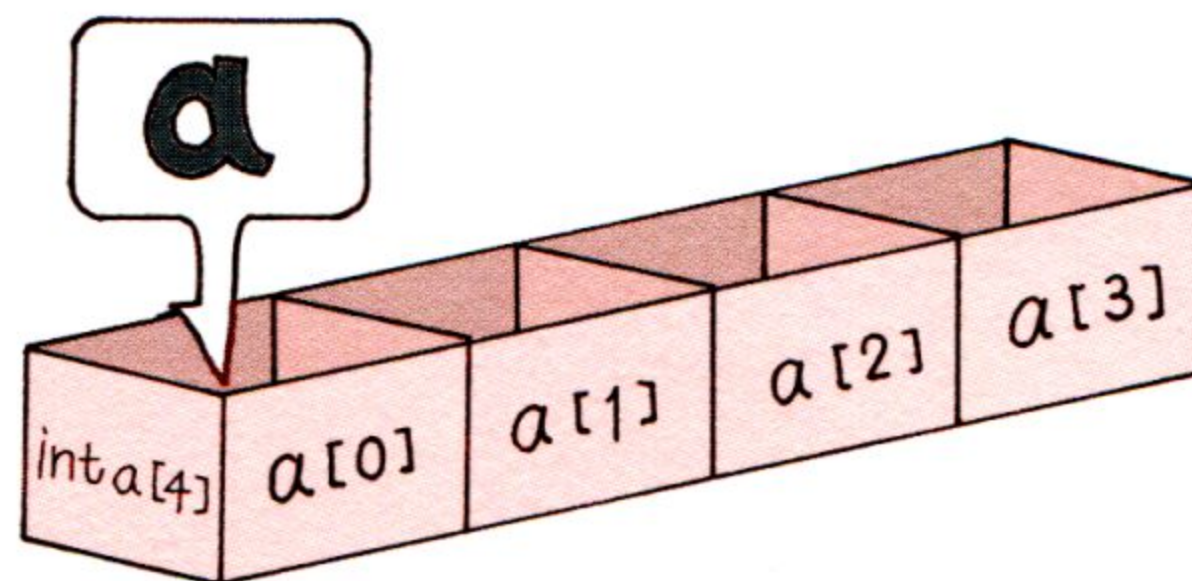
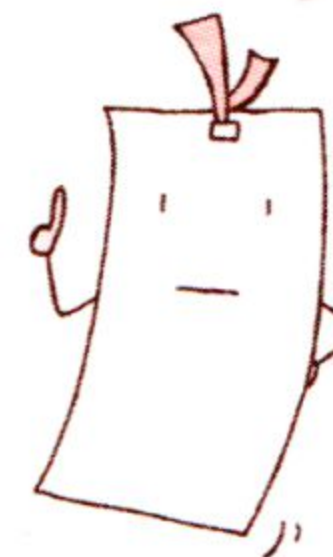
ポインタと配列

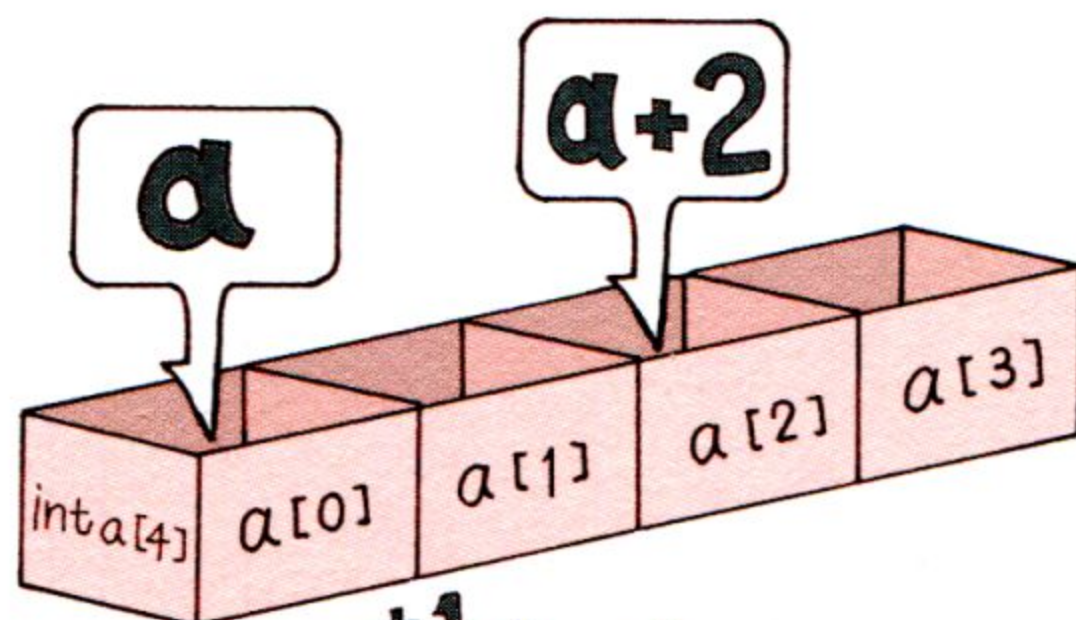
C言語では配列の名前とポインタは密接な関係があります。

ポインタと配列

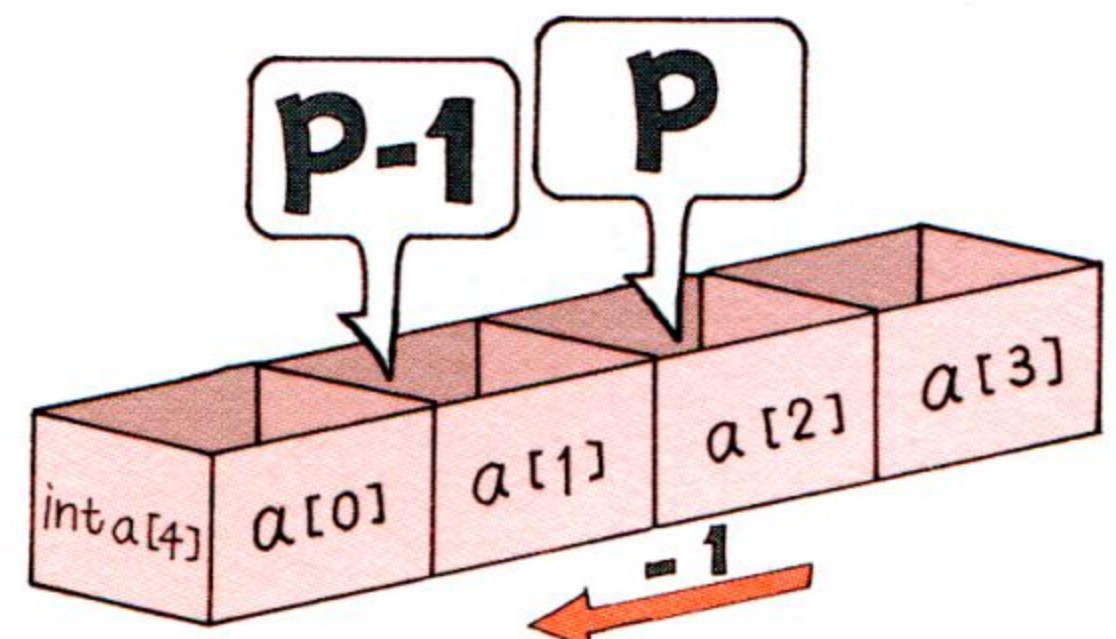
配列の名前そのものは、配列の最初の要素を指し示すポインタの役割をします。

```
int a[4];
```

…aはa[0]へのポインタを表します。

「&」（アドレスを得る記号）を使う必要はありません。

配列の最初の要素以降を呼び出すには、ポインタを加算していきます。ポインタに対しては整数の加算と減算のみ可能です。

```
int *p = a+2;
```


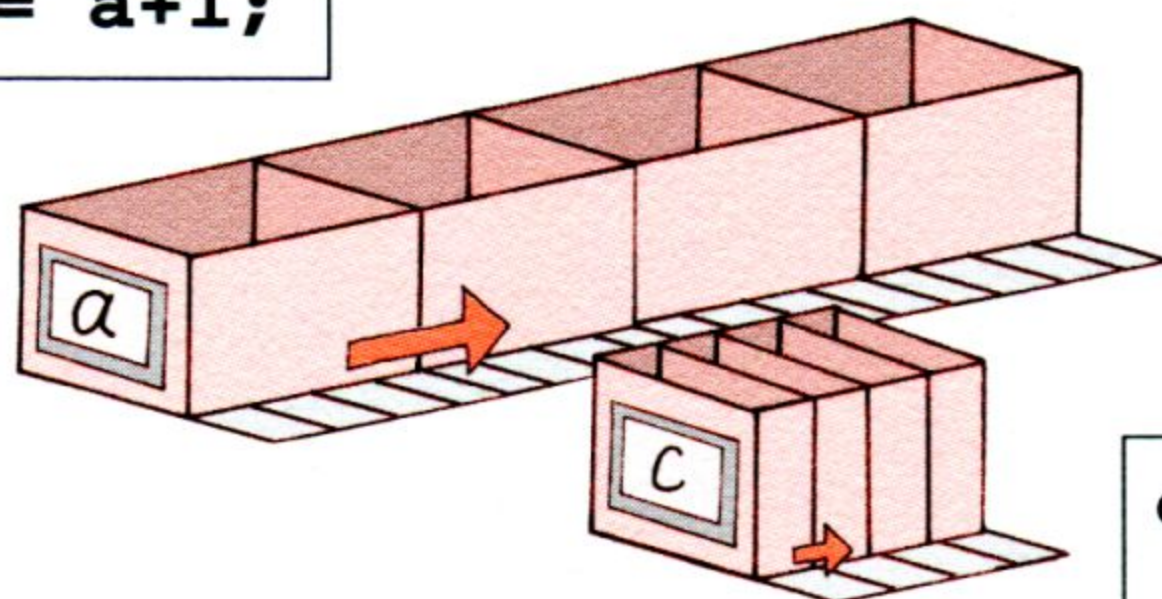
pはaから2つ先の箱a[2]を指します。

```
int *q = p-1;
```


qはpから1つ前の箱a[1]を指します。

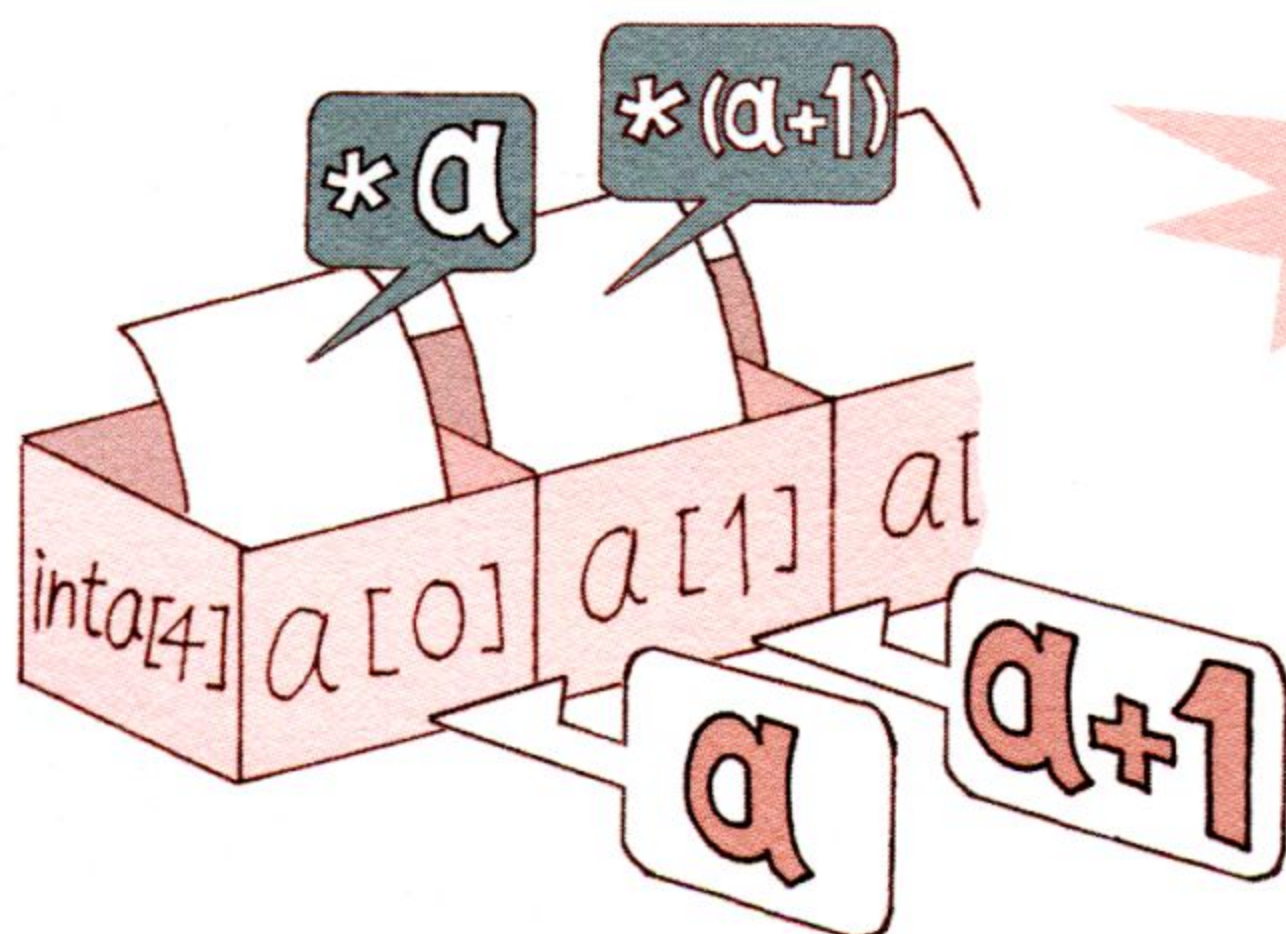
実際は配列の型によって、ポインタの進み方が違うことに注意してください。

```
long a[4];
long *p = a+1;
```

4バイト(=sizeof(long))ずつ進みます。

```
char c[4];
char *q = c+1;
```

1バイト(=sizeof(char))ずつ進みます。

C ポインタを使った配列の参照

配列aがあるとき、a自身は「a[0]へのポインタ」なので、*aは「aの格納場所にある値=a[0]」となります。同様に、a[1]=*(a+1)、a[2]=*(a+2)、…と書くこともできます。

注意 ( )をつけないと、違う意味になってしまいます。

***a+1**…a[0]の値に1を足す

例

```
#include <stdio.h>

main()
{
    int a[4] = {10, 20, 30, 40};
    printf("配列a[3]の値は%d¥n", *(a+3));
    printf("配列a[0]の値に3を足すと%d¥n", *a+3);
}
```

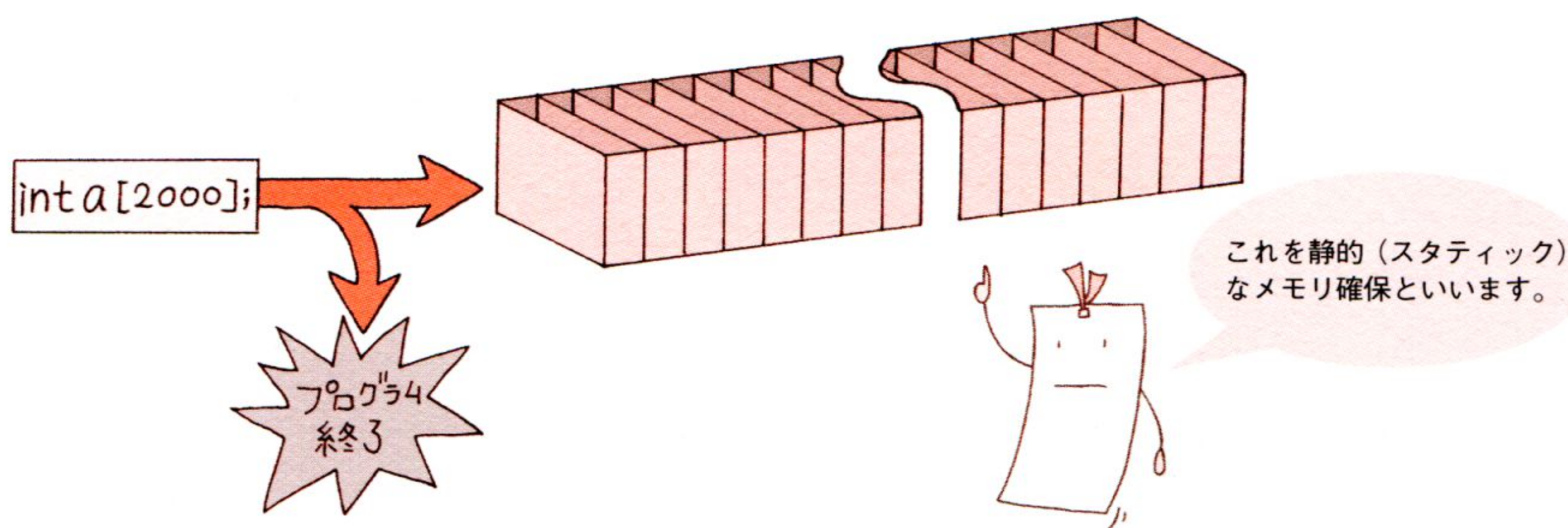
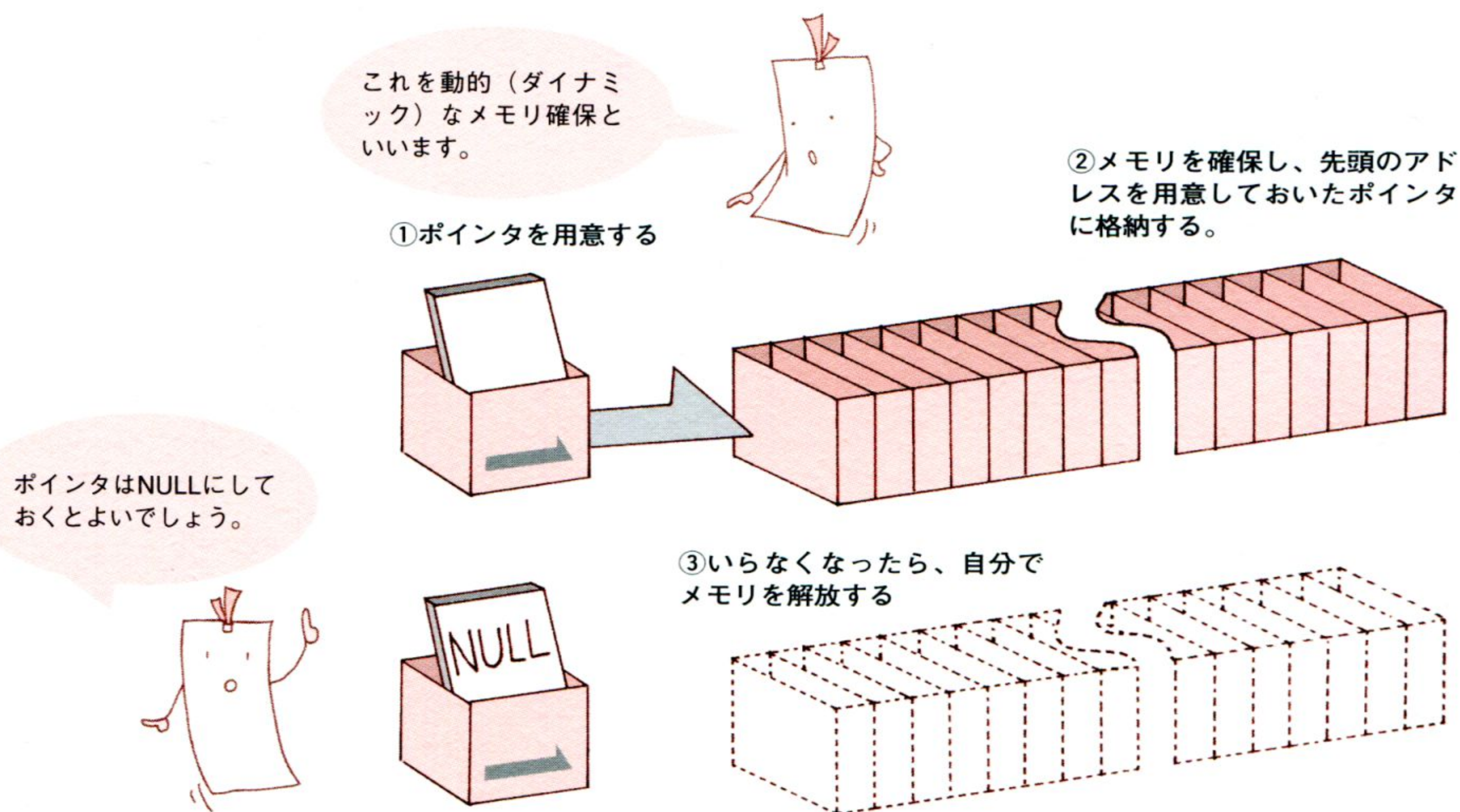
実行結果

```
配列a[3]の値は40
配列a[0]の値に3を足すと13
```

メモリの確保(1)

たくさんのメモリを使うときは、いきなり大きな配列を用意せず、プログラム中で用意した方が賢明です。

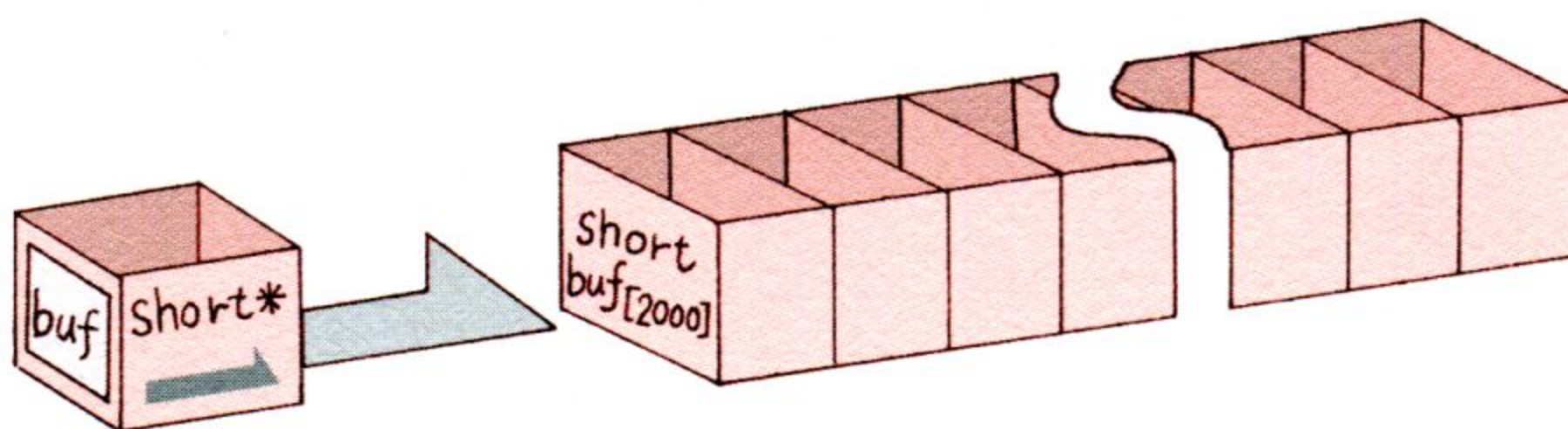
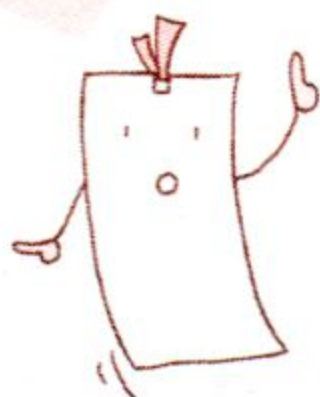
動的なメモリ確保

変数や配列を宣言すると、“自動的に”メモリ上にそれらの領域が確保されます。
しかし、この方法だと画像を扱うプログラムなど、たくさんのメモリを用意する必要がある場合、最悪、プログラムが止まってしまう危険性があります。

このようなときは、次のように“プログラムの処理として”メモリを確保します。

メモリ活用の手順

動的なメモリ確保を行うときに使う関数を紹介します。なお、これらの関数を使うときは、プログラムの先頭で、#include <memory.h>と#include <malloc.h>、および#include <stdlib.h>とする必要があります。

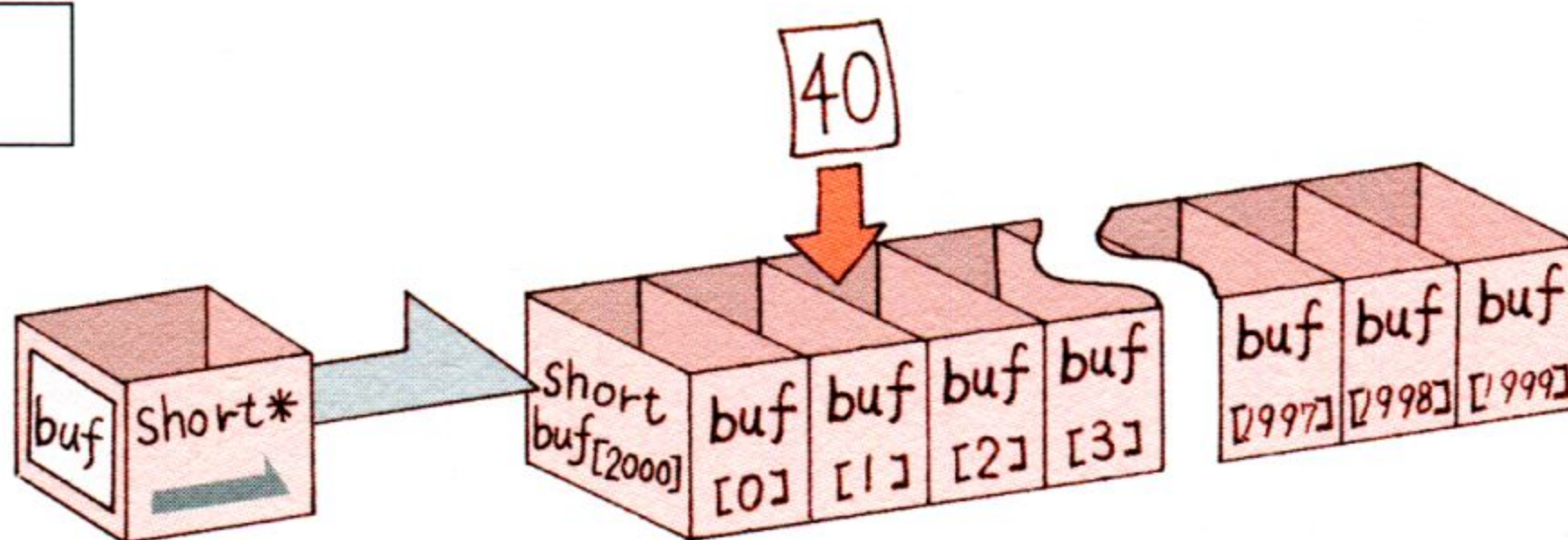
メモリの確保

メモリを確保し、用意しておいたポインタに先頭アドレスを格納します。

```
short *buf;
buf = (short *)malloc(sizeof(short)*2000);
```


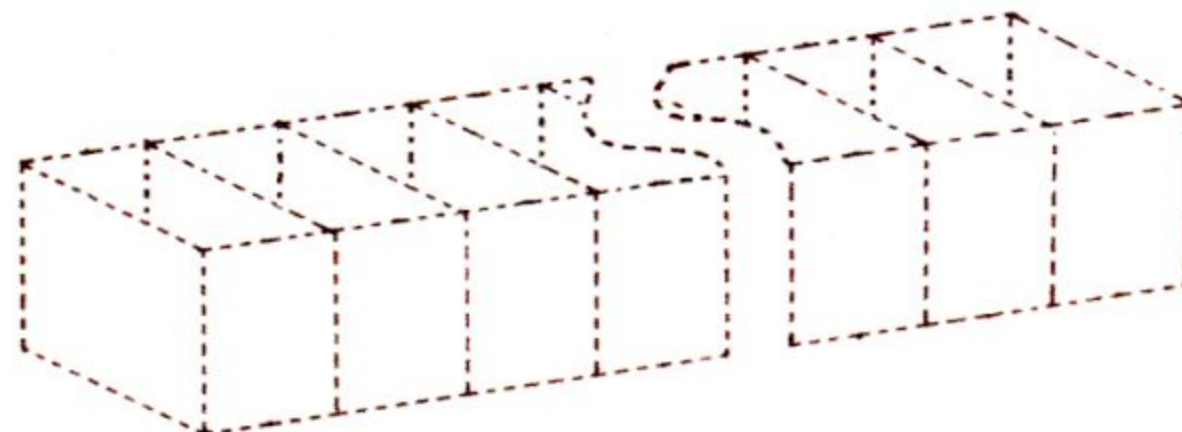
メモリの利用

確保してしまったあとは、通常の配列と同じように使えます。

```
buf[2] = 40;
```


メモリの解放

使い終わったら、メモリを解放します。

```
free(buf);
```

メモリの確保(2)

大量のメモリを扱うときに便利な関数を紹介します。

メモリ確保関係の関数

malloc()関数の代わりに、次のような関数を使うこともできます。

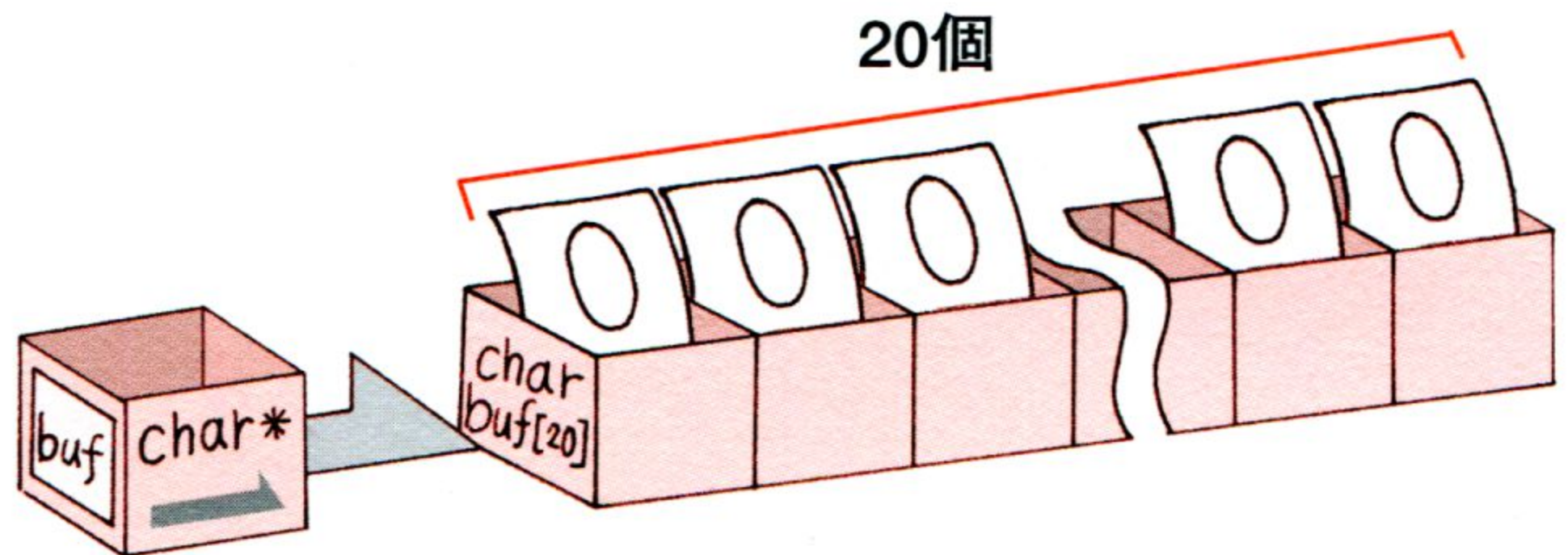
calloc()（キャロック） メモリを確保し、要素をすべて0に初期化する

```
buf = (char *)calloc(sizeof(char)*20);
```

確保したメモリの先頭アドレスを格納　　確保するメモリのバイト数

20個

buf char*

char buf[20]

realloc()（リアロック） 一度確保したメモリを違うサイズで確保し直す

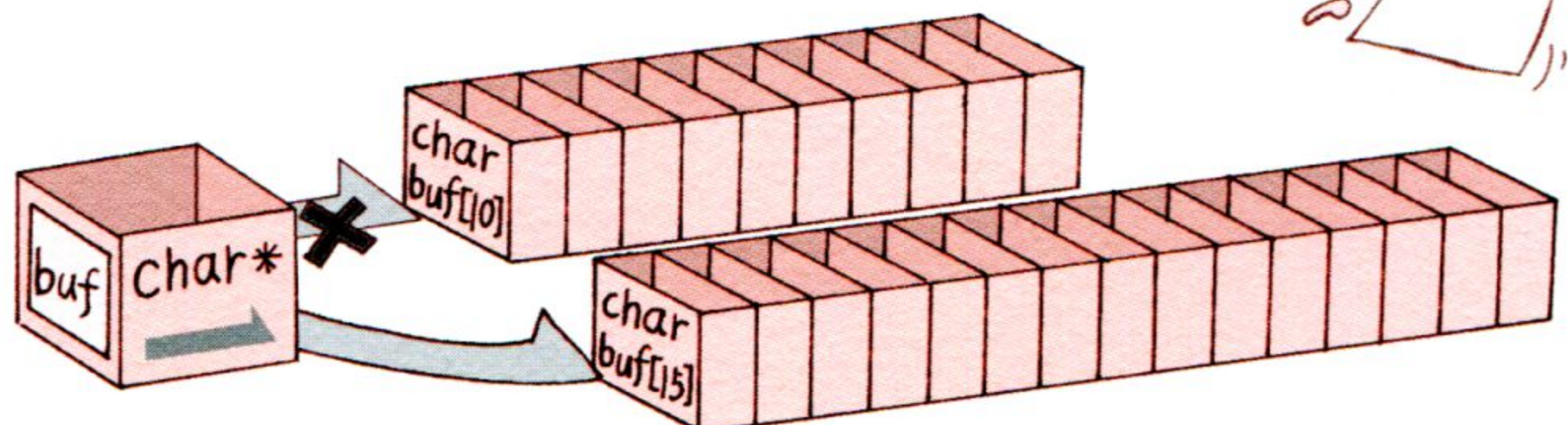
確保場所は変わってしまうかもしれません。

```
buf = (char *)realloc(buf, sizeof(char)*15);
```

新しいメモリ領域の先頭アドレスを格納　　以前のメモリ領域の先頭アドレス　　新しく確保するメモリのバイト数

buf char*

char buf[10]

char buf[15]

C メモリ操作関数

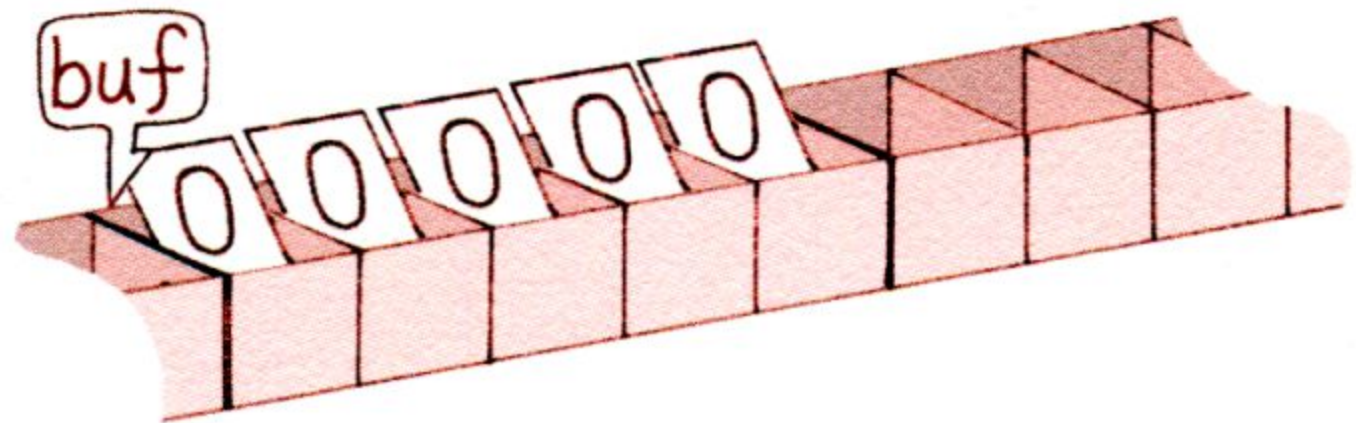
メモリの内容を操作するときに便利な関数を紹介します。なお、これらの関数を使うときは、プログラムの先頭で、#include <memory.h>とする必要があります。

memset( ) メモリの内容をすべて同じ値に設定する

（メムセット）

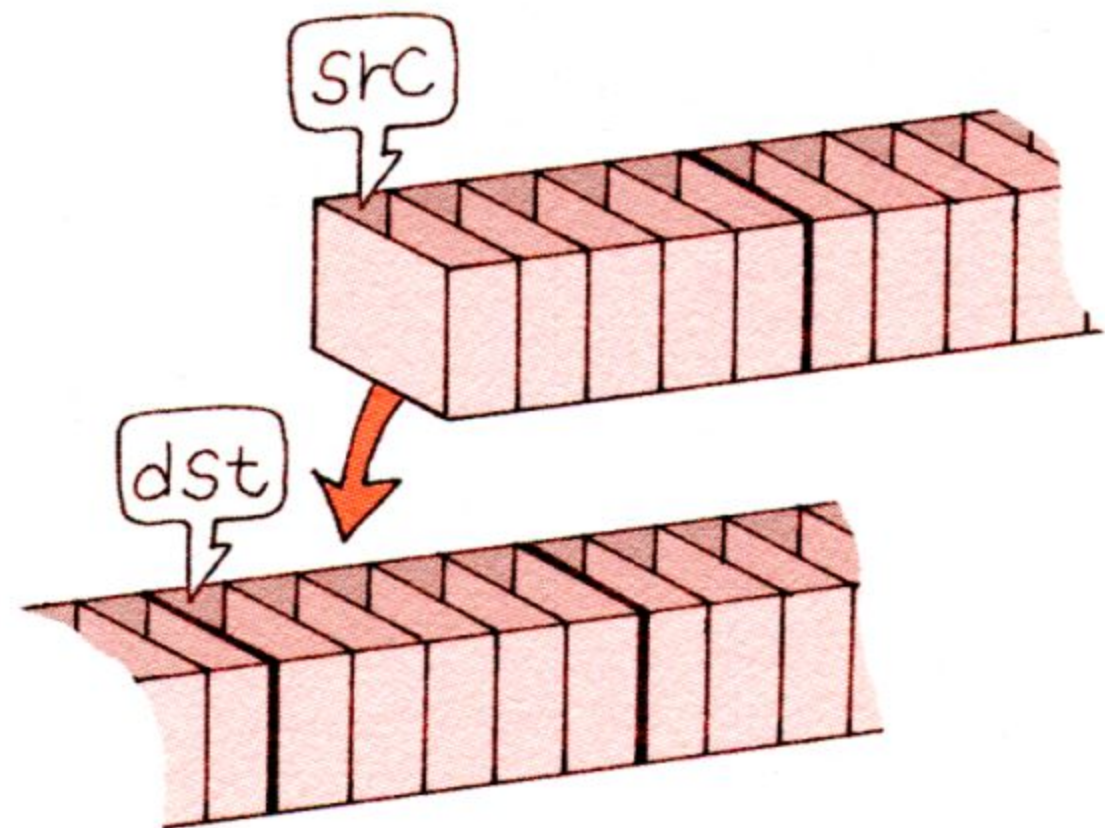
```
memset(buf, 0, 5);
```

- buf：メモリ領域の先頭アドレス
- 0：設定する値
- 5：値を設定するメモリのバイト数

memcpy( ) メモリの内容をコピーする

（メムコピー）

```
memcpy(dst, src, 5);
```

- dst：コピー先メモリ領域の先頭アドレス
- src：コピー元メモリ領域の先頭アドレス
- 5：コピーするメモリのバイト数

例

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>

main()
{
  char *b;
  char a[4] = {20, 40, 30, 10};
  b = (char *)malloc(sizeof(char)*200);
  if(!b)
    return;
  memcpy(b, a, sizeof(char)*4);
  printf("%d %d %d %d\n", b[0],b[1],b[2],b[3]);
    free(b);
}
```

if(!b) return; ：メモリが確保できなかったときのチェックを必ず行います。

return; ：main( )関数を終了し、何も処理しません。

実行結果

```
20 40 30 10
```

サンプルプログラム

■文字列の中から文字を探す

77ページで登場したstrchr()関数のようなものを作ってみます。ただし、最初に登場した位置だけでなく、すべての位置を表示させています。

ソースコード

```
#include <stdio.h>
main()
{
    char s[] = "I love cat and dog."; /* 探す対象の文字列 */
    char c = 'a'; /* 探す文字 */
    char *p = s;
    int n = 0;

    printf("¥"%s¥"の中から¥'%c¥'を探します。¥n", s, c);
    while(*p != '¥0') {
        if(*p == c) {
            printf("%d文字目で発見しました。¥n", p-s+1);
※           n++;
        }
        p++;
    }
    if(n == 0)
        printf("1つも見つかりませんでした。¥n");
    else
        printf("全部で%d個見つかりました。¥n", n);
}
```

実行結果

```
"I love cat and dog."の中から'a'を探します。
9文字目で発見しました。
12文字目で発見しました。
全部で2個見つかりました。
```

同じプログラムをstrchr()関数を使って作るときは、※の部分を次のよう変えます（先頭に「#include <string.h>」を追加してください)。

```
    while(1) {
        p = strchr(p, c);
        if(!p)
           break;
        printf("%d文字目で発見しました。¥n", p-s+1);
        n++;
        p++;   ← 探しあてた文字の次の位置から探します。
    }
```

■表計算

4列×3行となっている2次元配列の縦の列（4列）、横の列（3行）、すべての値の合計を表示します。難しい処理はありませんが、どれとどれが対応するのか混乱しないようにしてください。

ソースコード

```
#include <stdio.h>

main()
{
    int mat[3][4] = {
        {20, 42, 70, 34},
        {67, 98, 37, 41},
        {76, 99, 43, 65}
    };
    int i, j;
    int sum_r; /* 横の列の和 */
    int sum_c[4] = {0, 0, 0, 0}; /* 縦の各列の和 */
    int total = 0; /* 全ての数の合計 */

    /* 各要素の表示と計算 */
    for(j = 0; j < 3; j++) {
        sum_r = 0;
        for(i = 0; i < 4; i++) {
            printf("%4d ", mat[j][i]);
            sum_r += mat[j][i];
            sum_c[i] += mat[j][i];
        }
        printf("| %4d¥n", sum_r);
    }

    /* 仕切り線と最後の行の表示 */
    printf("--------------------+-----¥n");
    for(i = 0; i < 4; i++) {
        printf("%4d ", sum_c[i]);
        total += sum_c[i];
    }
    printf("| %4d¥n", total);
}
```

表のデータ

横の合計は表示順に求められるので配列にはしません。

|, -, +は仕切り用の文字です。

実行結果

```
  20   42   70   34 | 166
  67   98   37   41 | 243
  76   99   43   65 | 283
--------------------+--------
 163  239  150  140 | 692
```

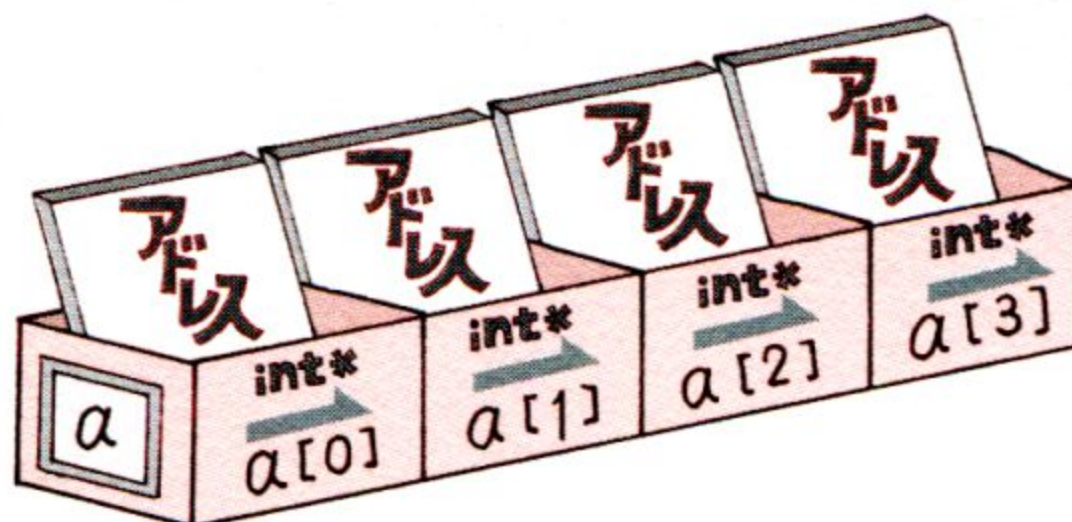
COLUMN コラム

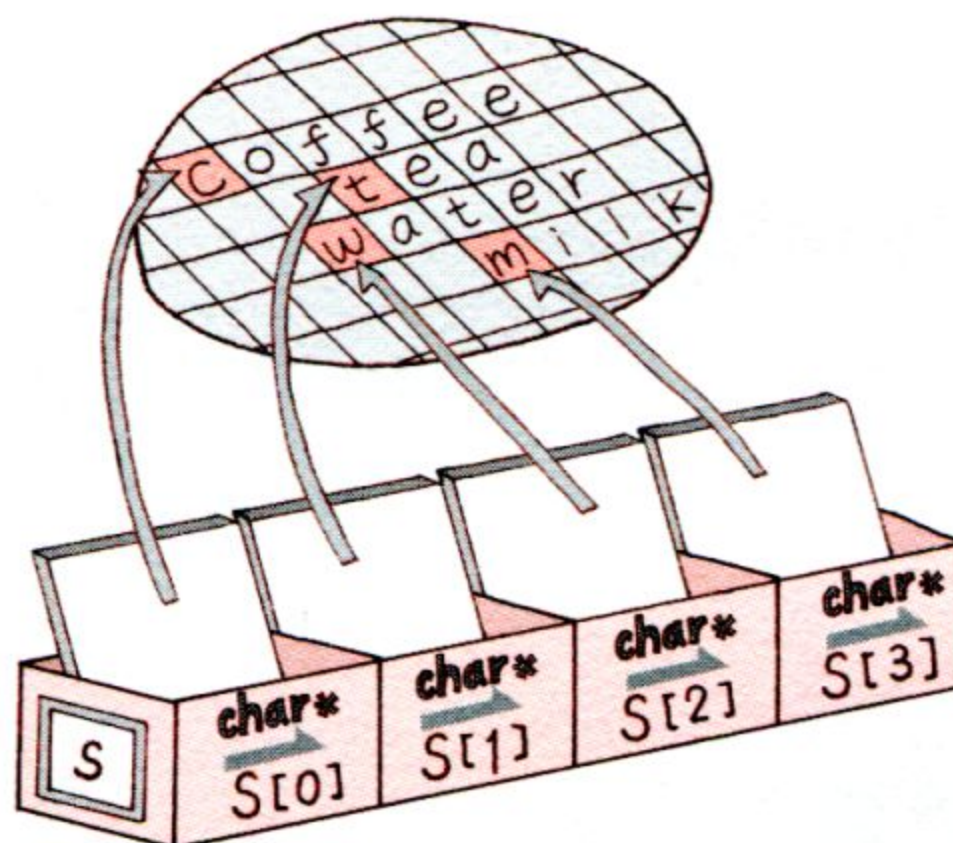
～ポインタ配列～

ポインタと配列の応用編として、各要素がポインタ（値がアドレス）であるような配列を考えてみましょう。このような配列のことをポインタ配列といいます。

ポインタ配列は次のように宣言します。

```
int *a[4];
```

要素数

ポインタ配列の初期化は次のように行います。

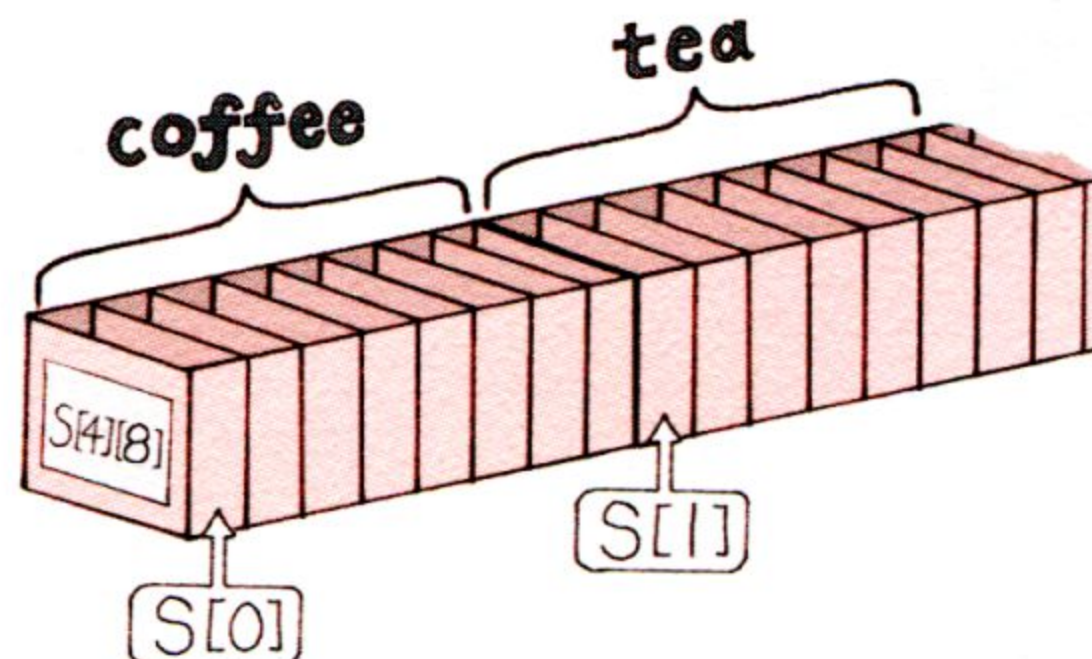
```
char *s[4];
s[0]="coffee";
s[1]="tea";
s[2]="water"
s[3]="milk"
```

この場合、メモリ上にcoffee、tea、water、milkという文字列データが作られて、ポインタ配列s[4]の要素にはそれぞれの最初の文字'c'、't'、'w'、'm'のある場所のアドレスが入ります。

これに対し、多次元配列で4つの文字列を格納すると次のようになります。メモリの使い方がまったく異なることに注意してください。

```
char s[4][8] = {
    "coffee",
    "tea",
    "water",
    "milk"
};
```

5 関数

第5章

第5章は ここがkey!

Topics 魔法のブラックボックス

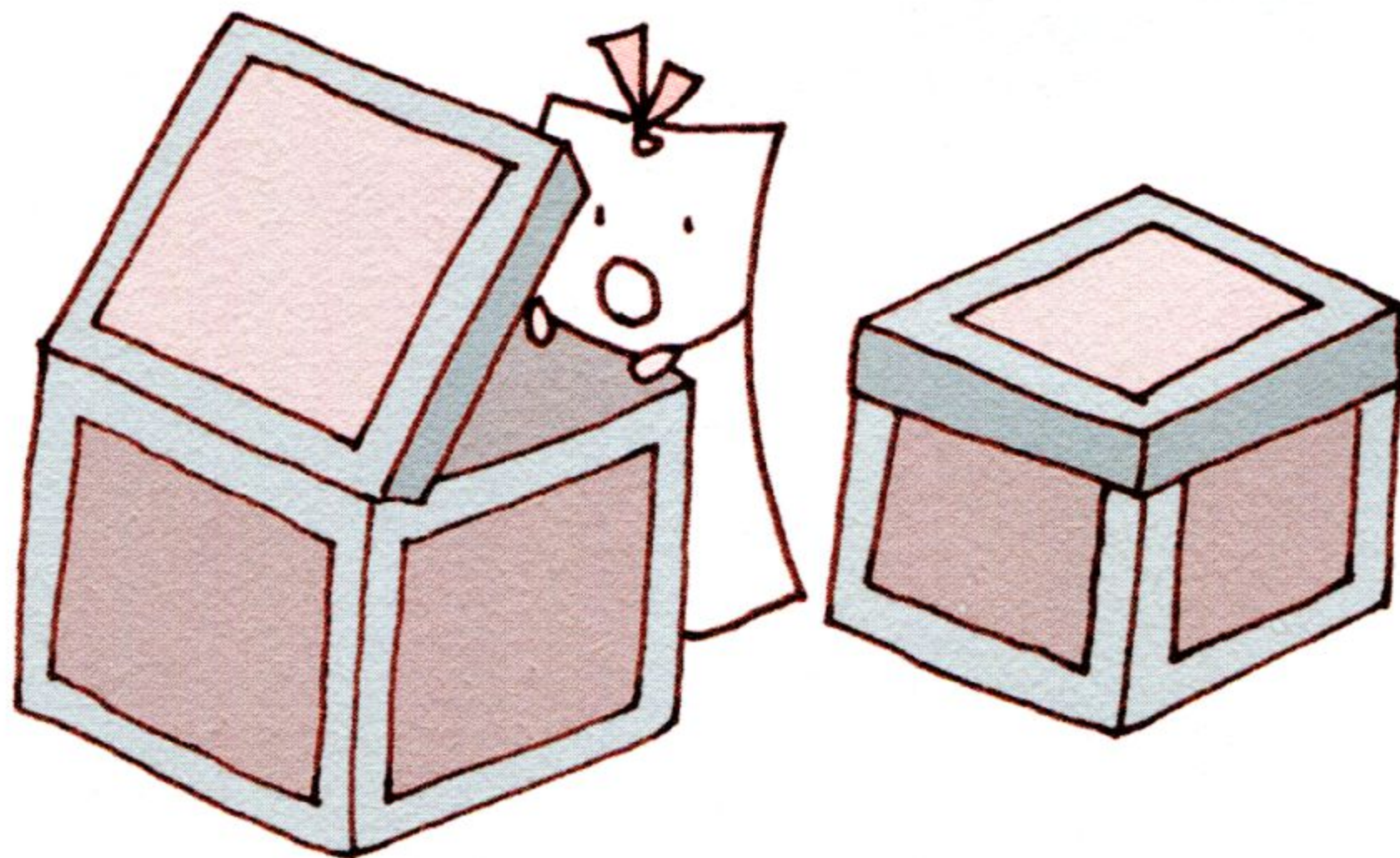
この章では、関数について学んでいきます。第１章の冒頭で少しふれたとおり、C言語の関数は、「一連の処理の集り」です。たとえば、次のように、printf()関数を実行したとします。%xは、データの書式を16進表記に変換するための書式指定です。

```
printf("%x¥n", 10);
```

コンピュータの画面には16進数のaが表示されます。一見簡単なことのように思えますが、関数を呼び出してから画面に文字を表示するまでの間に、printf()関数の内部では、書式指定を分析して、書式を変換し、画面に出力する、という一連の処理を実行しているのです。

このように、関数を利用すると、面倒な処理を記述することなく、いろいろな機能を実現することができます。いってみれば、関数は、大変便利な魔法のブラックボックスなのです。これを利用しない手はありませんね。

関数という言葉にとっつきにくいイメージや、苦手意識を持っている人も、これで少しは興味がわいてきたでしょうか。

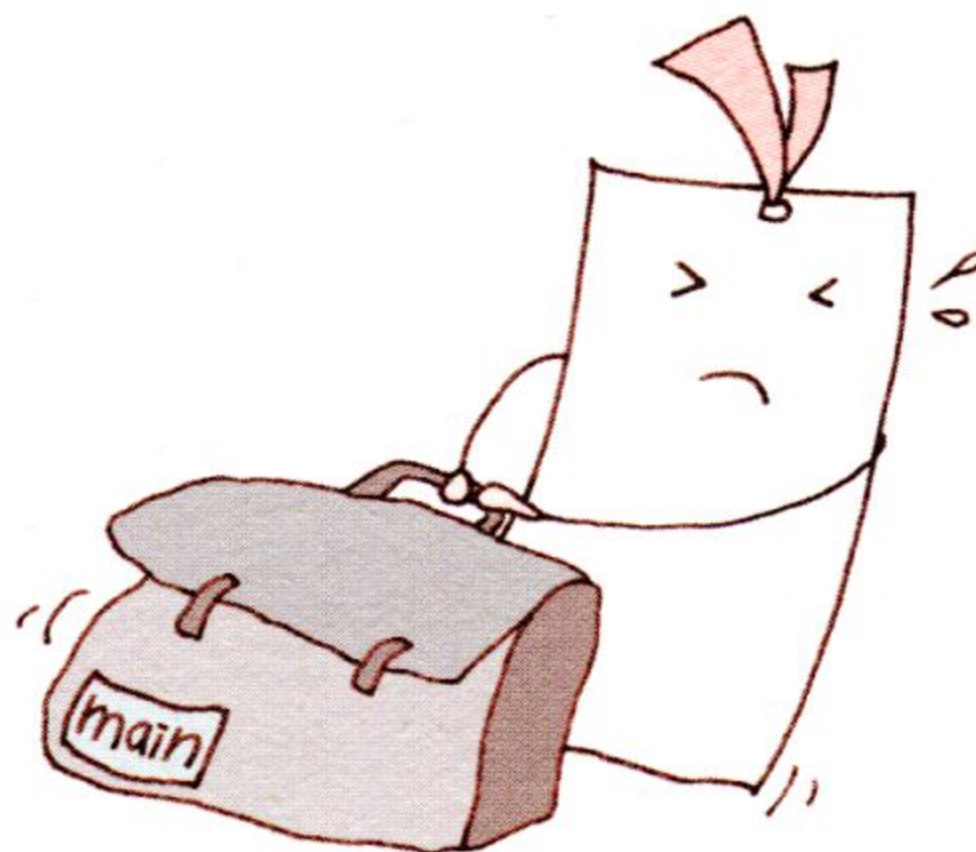
Topics

実践的なプログラムへの第一歩

printf()、strcpy()など、C言語があらかじめ用意している関数（**標準ライブラリ関数**）の他に、プログラマが独自の関数を作ることもできます。

今度は、ブラックボックスの中身を自分で作ることになりますから、少し手間がかります。この章では、関数を作り、利用するうえで必要な知識を、基礎から詳しく説明していきます。具体的には、変数の**有効範囲**や、関数のひな型をコンパイラに知らせる**プロトタイプ**、また、関数への**データの渡し方**、といったことを学んでいきます。

なんだか面倒くさそうですね。自分で関数など作りたくなくなってしまいます。でも、もし関数を作らなかったら、main()関数の中に、何十行、何百行のソースコードを書くことになります。これは、大きな旅行かばんの中に、歯ブラシやら財布やら、すべての荷物をそのまま詰め込むのと同じことです。これでは、いざ何かを使いたいときに、なかなか目的のものを見つけられないでしょう。

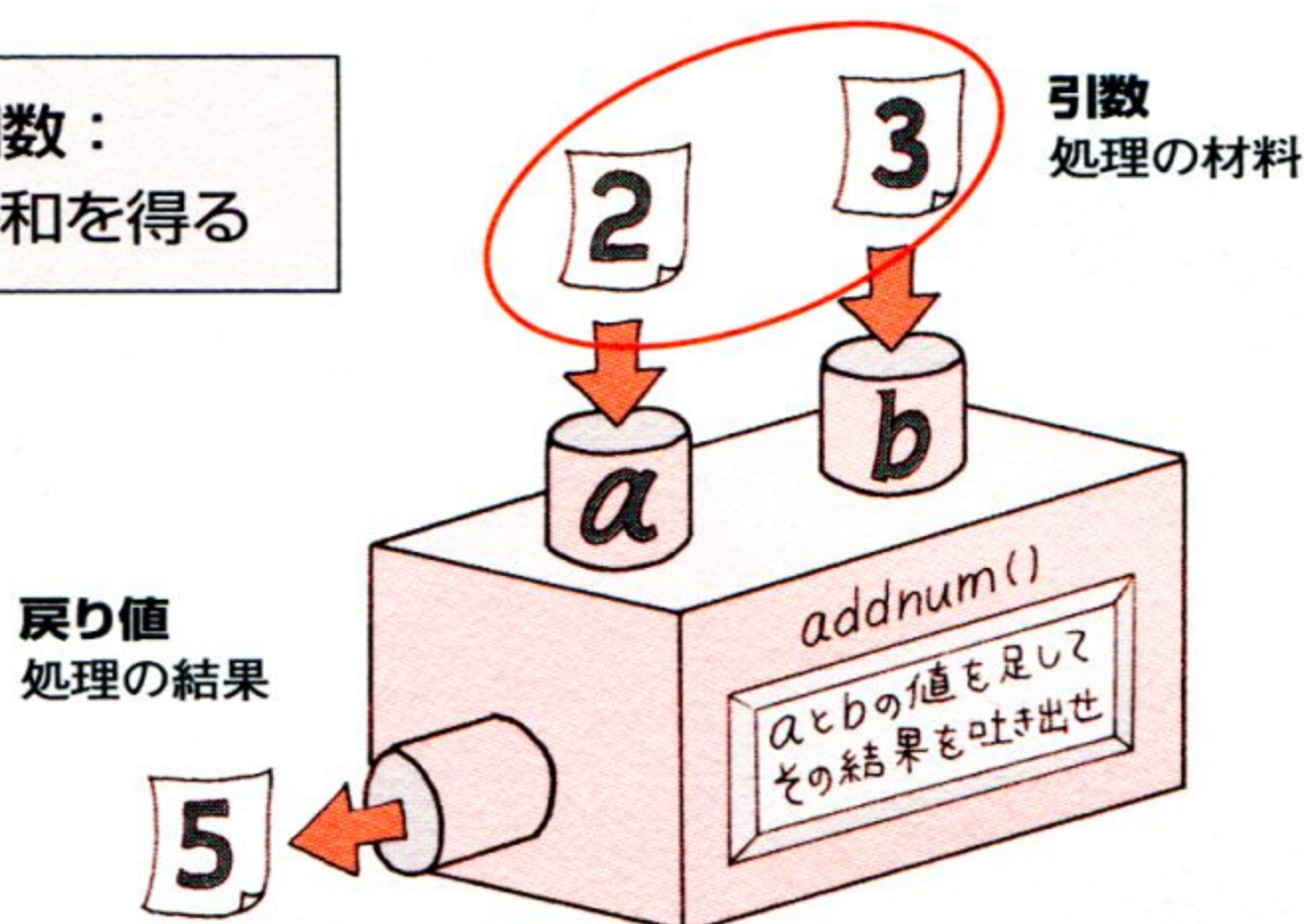
実際は、洗面道具や衣類、貴重品など、目的や用途によって必要なものをまとめて、それからかばんの中に入れますよね。プログラムを書くときも、処理ごとにいくつかの関数にまとめ、それらの関数をmain()関数の中から呼び出す方がすっきりします。

関数を理解することは、実践的なプログラムへの第一歩です。ゆっくりでかまいませんから、この章の内容をしっかりと理解し、関数を自由自在に使いこなせるようになりましょう。

関数の定義

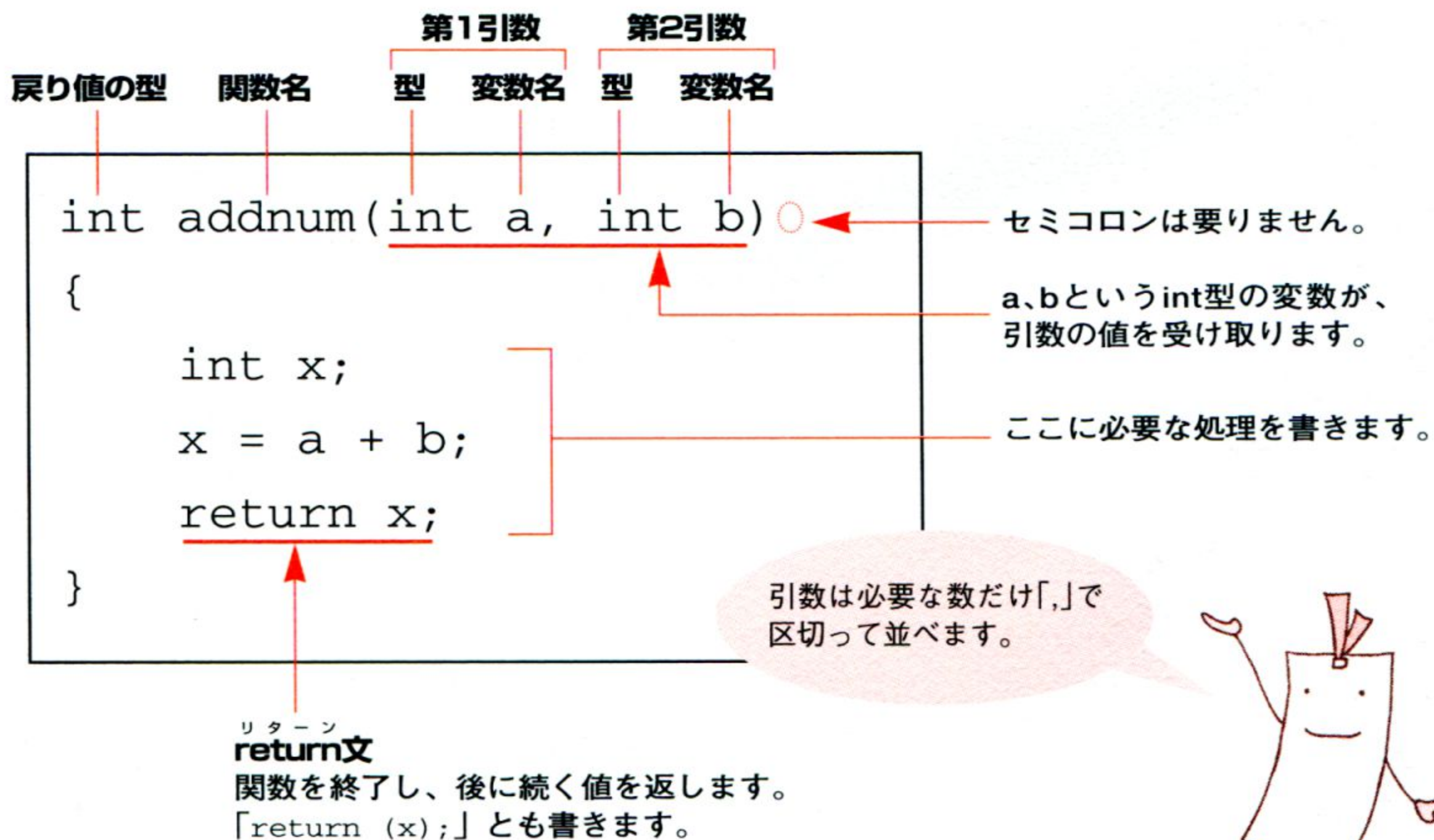
関数の概念を理解し、C言語で関数を定義する方法を見ていきましょう。

関数とは？

関数とは、プログラマが与えた値を指示どおりに処理し、結果を吐き出す箱のようなものです。処理の材料となる値のことを**引数**（ひきすう）（**パラメータ**）といい、結果の値のことを**戻り値**（**返り値**）といいます。たとえば、次のような関数を考えてみましょう。

addnum()関数：
2つの整数値の和を得る

上の関数をC言語で記述すると、次のようになります。このように、関数の機能を記述することを「関数を**定義する**」といいます。


```
int addnum(int a, int b)
{
    int x;
    x = a + b;
    return x;
}
```

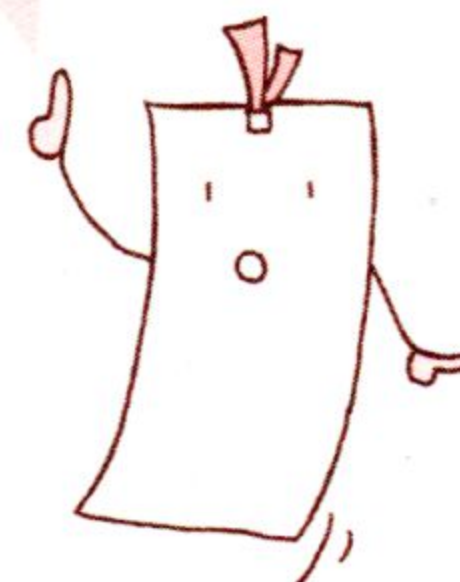

return文（リターン）
関数を終了し、後に続く値を返します。
「return (x);」とも書きます。

戻り値や引数をもたない関数

関数が値を返す必要がないときは、戻り値の型にvoid（ボイド）と指定します。次のような関数を考えてみます。

voidは「空の」という意味です。

dispnum()関数：引数の整数値を表示する

```
void dispnum(int a)
{
    printf("引数の値は%d¥n", a);
    return;
}
```

戻り値を指定する必要はありません（この場合、return文がなくてもかまいません）。

また、引数が必要ないときには、次のように関数を定義します。

hello()関数：「Hello World」と表示する

```
void hello(void)
{
    printf("Hello World¥n");
}
```

「void hello()」とも書きます。

標準ライブラリ関数

printf()やstrcpy()のような、C言語があらかじめ用意している関数のことを標準ライブラリ関数といいます。これらの関数の定義は、プログラムが動作する環境の中にあります。プログラマは定義しなくてもこれらの関数を利用することができます（詳しくは第8章参照）。

プログラマが書いたプログラム

```
#include <stdio.h>
main()
{
        ⋮
    printf("Hello¥n");
        ⋮
}
```

＋ 標準ライブラリ（関数の定義） → 実行ファイル

関数の呼び出し

定義した関数を呼び出し、実行する方法を見ていきましょう。

関数呼び出しの基本

関数の定義に対し、呼び出し部分の書き方は次のようになります。

関数の定義

```
void dispnum(int a)
{
      ⋮
}
```

aの値は5になります。

対応

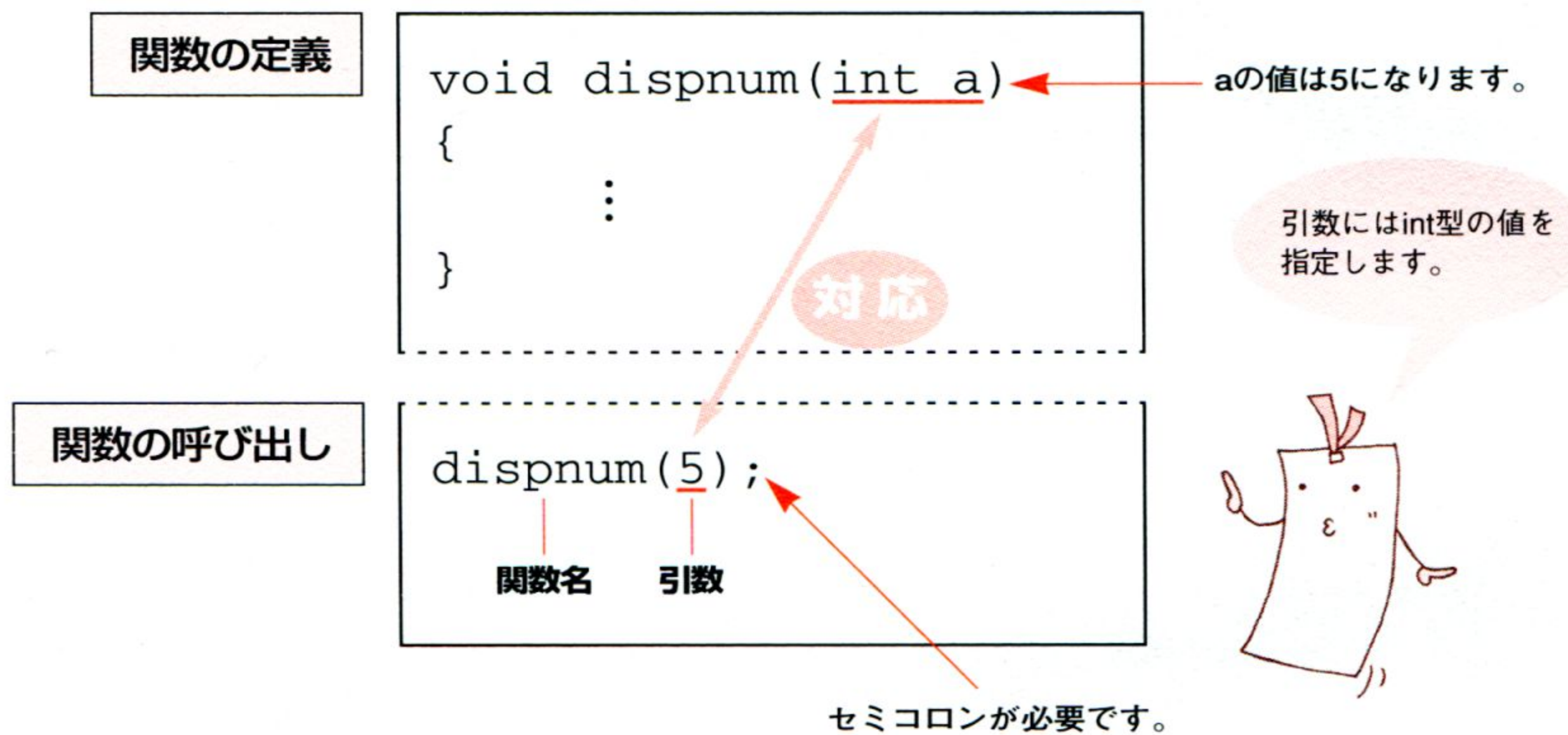
関数の呼び出し

```
dispnum(5);
```

関数名　引数

セミコロンが必要です。

引数にはint型の値を指定します。

例

```
#include <stdio.h>

void dispnum(int a)
{
    printf("引数の値は%d¥n", a);
}

main()
{
    int x = 10;

    dispnum(5);
    dispnum(x);
}
```

5を引数として、dispnum()関数を実行します。

xの値10を引数として、dispnum()関数を実行します。

実行結果

```
引数の値は5
引数の値は10
```

戻り値を利用する

関数が値を返すときは、戻り値の型に応じた変数を用意して、その中に結果を代入します。

関数の定義

```
int addnum(int a, int b)
{
        ⋮
}
```

対応

関数の呼び出し

```
int n;
n = addnum(2, 3);
```

関数の戻り値をnに代入します。

戻り値を格納する変数の型は、関数の戻り値の型と同じにします。

例

```
#include <stdio.h>

int addnum(int a, int b)
{
    int x;

    x = a + b;
    return x;
}

main()
{
    int n;

    n = addnum(2, 3);
    printf("戻り値は%d¥n", n);
}
```

addnum()関数を実行し、戻り値をnに格納します。

実行結果

```
戻り値は5
```

変数のスコープ

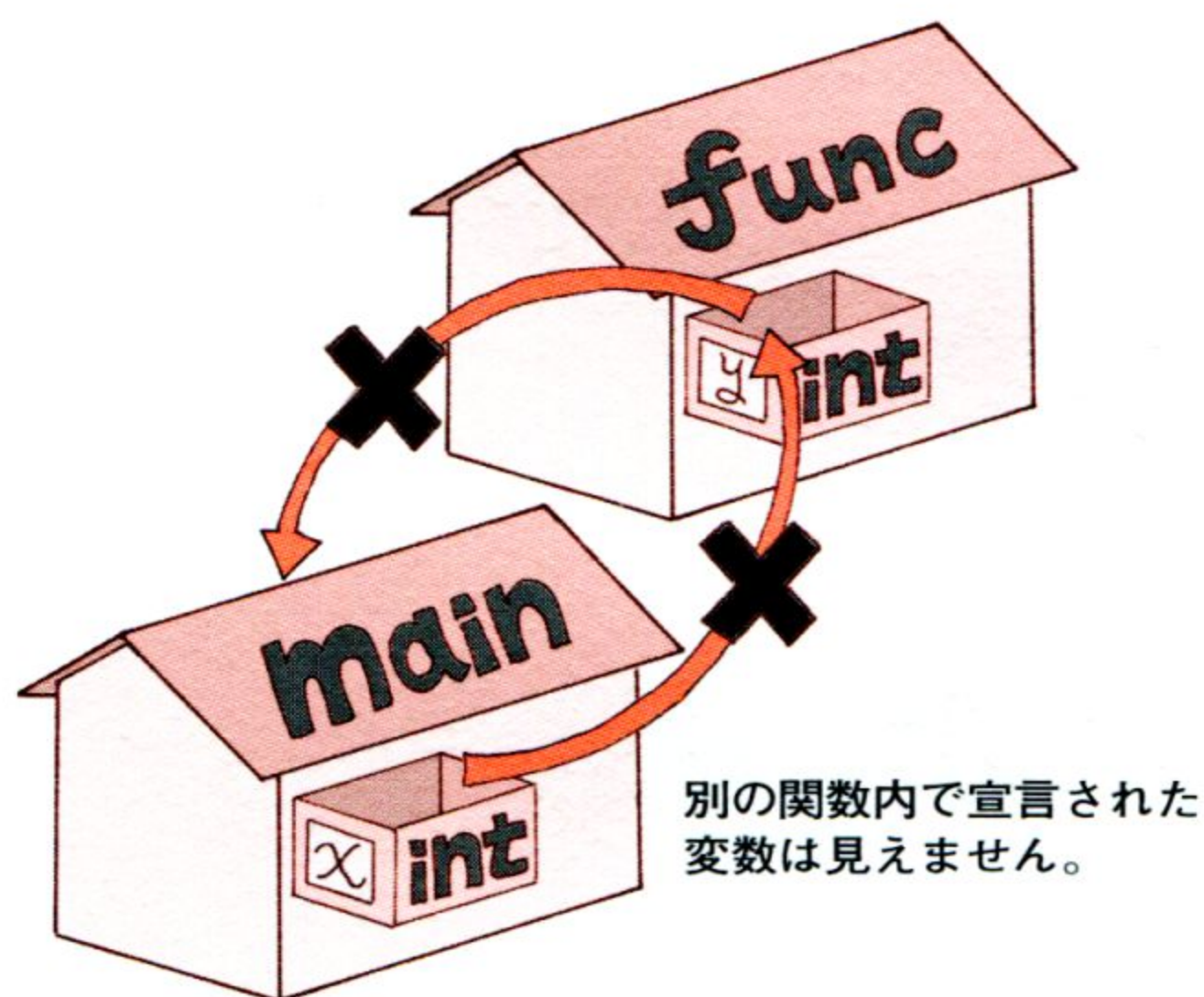
変数を宣言する場所によって、変数の有効範囲が異なります。

ローカル変数とグローバル変数

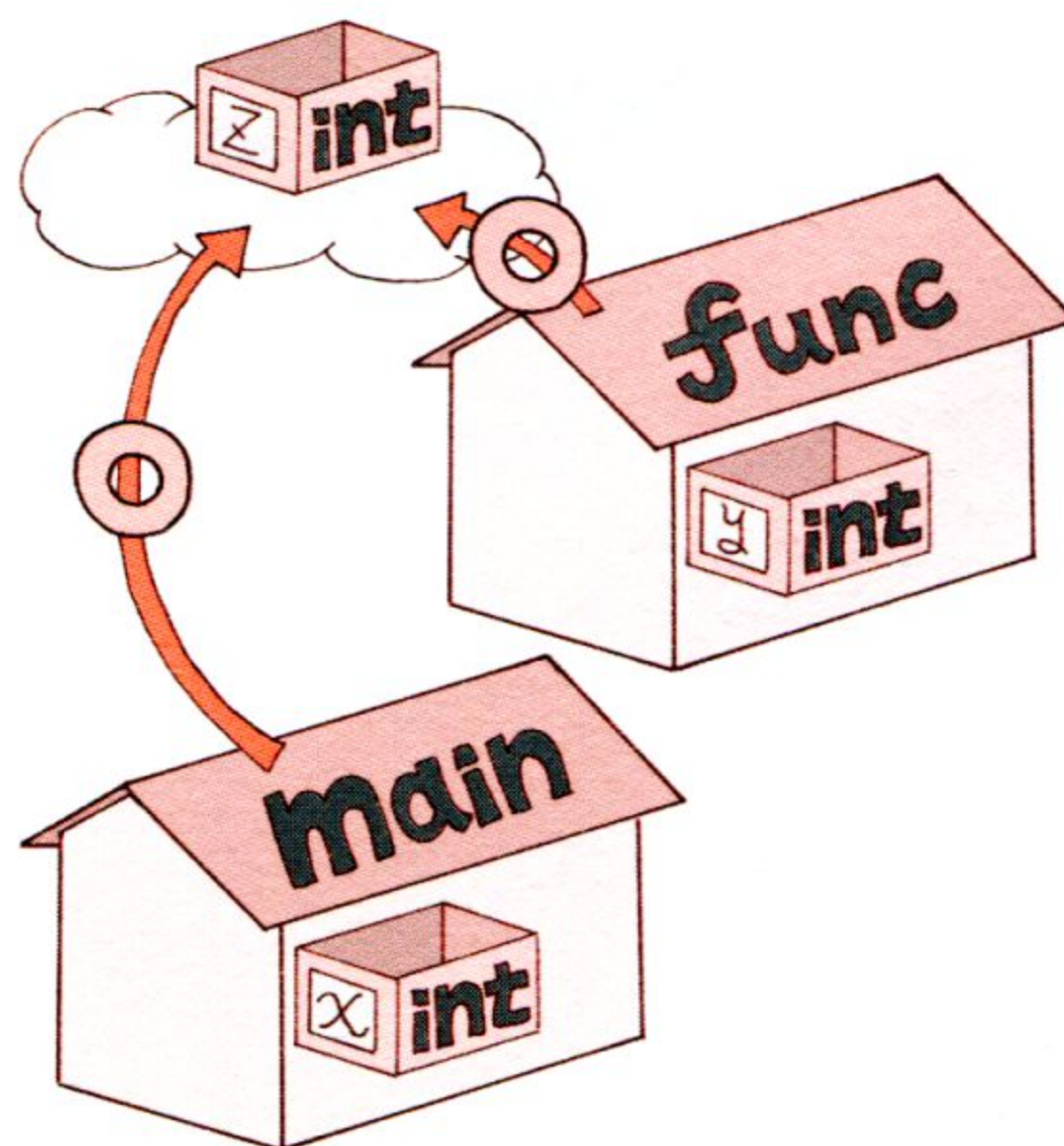
関数の中で宣言した変数のことを**ローカル変数**といいます。ローカル変数を参照できる範囲は、変数を宣言した関数の内側に限られます。変数の有効範囲のことを、変数の**スコープ**といいます。

```
void func()
{
        int y;     変数yのスコープ
         :
}
```

```
main()
{
        int x      変数xのスコープ
        x = 3:
        y = 5;
         :
}
```

func()にあるyを参照することはできません。

関数の外で宣言した変数のことを**グローバル変数**といいます。グローバル変数は、変数の宣言以降に定義したすべての関数から参照できます。

```
int z;              変数zのスコープ
void func(...)
{
        int y;      変数yのスコープ
        z = 2;
         :
}
```

```
main()
{
        int x       変数xのスコープ
        z = 1:
         :
}
```

例

```
#include <stdio.h>

int y;
int z;

void myfunc(int a)
{
        int z;
        int x;
        x = a;
        y = a;
        z = a;
}

main()
{
        int x;
        x = 10;
        y = 10;
        z = 10;
        printf("x,y,zの値は%d,%d,%d¥n", x, y, z);
        myfunc(5);
        printf("x,y,zの値は%d,%d,%d¥n", x, y, z);
}
```

グローバル変数y、zのスコープ

グローバル変数と同じ名前のローカル変数があるときは、ローカル変数が優先です。

ローカル変数x、zのスコープ

同じ名前のローカル変数どうしは、別の変数とみなします。

ローカル変数xのスコープ

実行結果

```
x,y,zの値は10,10,10
x,y,zの値は10,5,10
```

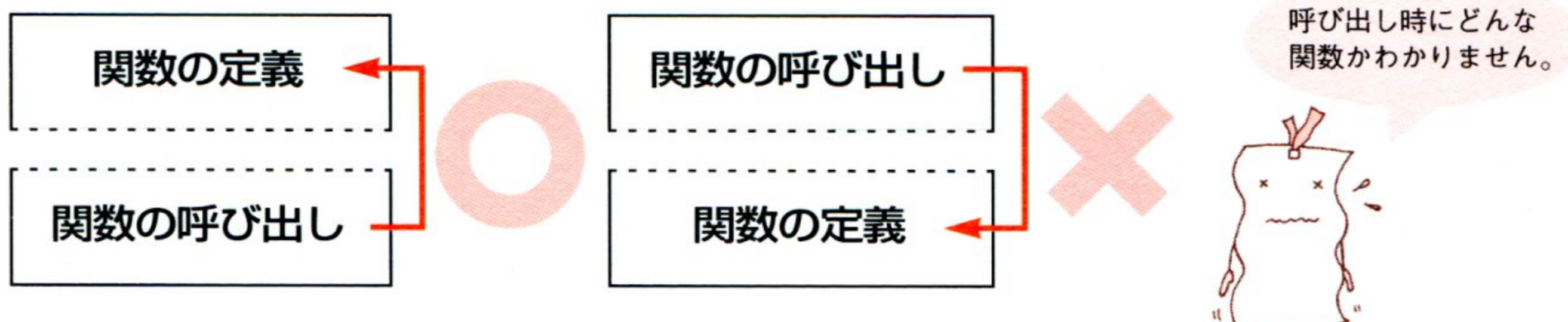
グローバル変数yの値だけが、変化します。

関数myfunc( )の中では、xとzはローカル変数として、yはグローバル変数として扱われたわけです。

プロトタイプ

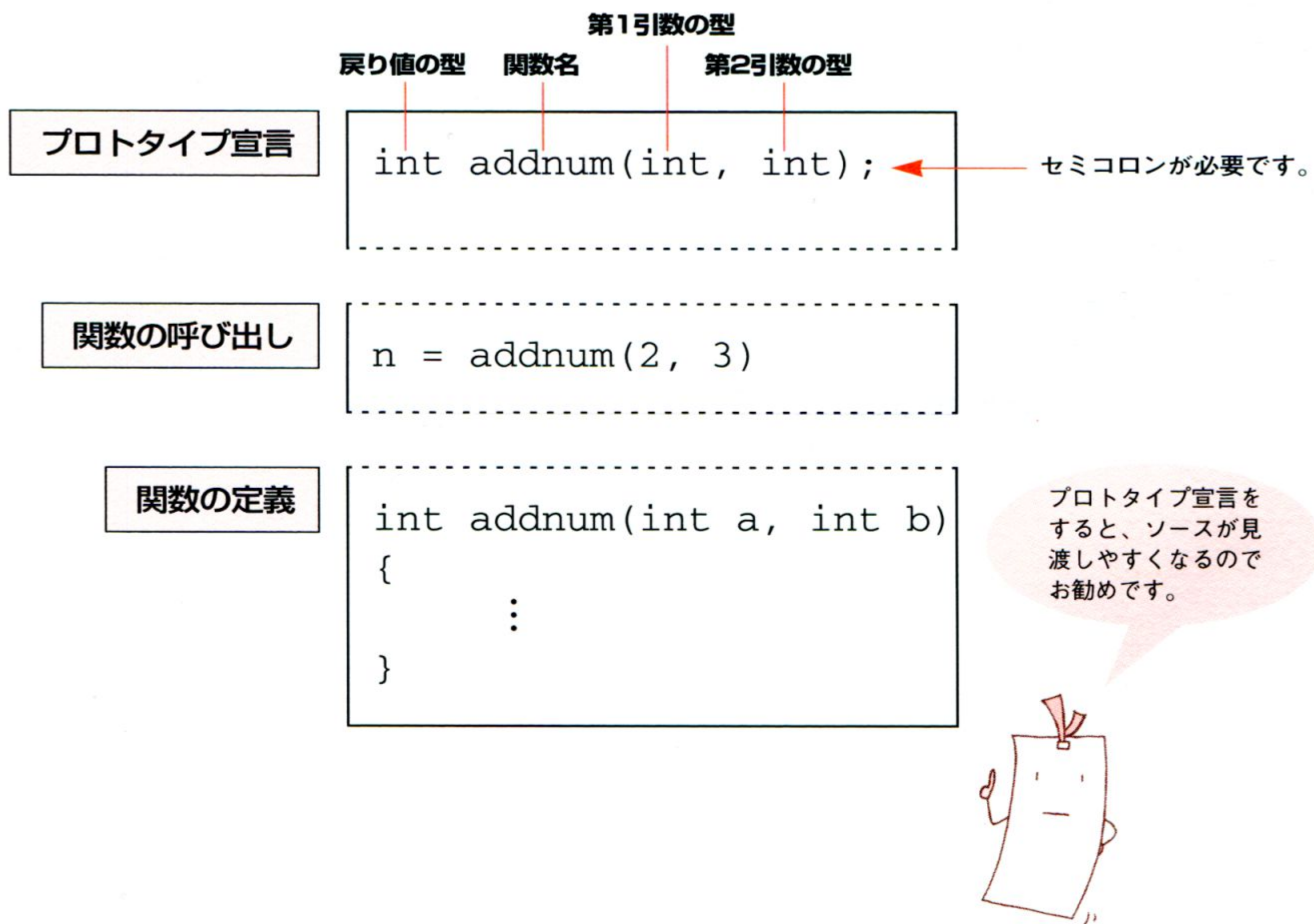
関数を呼び出す前には、関数のプロトタイプ（ひな型）を宣言します。

関数のプロトタイプを宣言する

今までは「関数の定義」→「関数の呼び出し（main()関数）」の順番でコーディングしてきました。もし、これを逆にするとコンパイルエラーになる場合があります。

そんなときは、プロトタイプという関数のひな型を呼び出し前に宣言しておきます。
プロトタイプ宣言は、関数の仕様にあたる部分だけを抜き出したものです。


```
int addnum(int, int);
```

```
n = addnum(2, 3)
```

```
int addnum(int a, int b)
{
        ⋮
}
```

関数の形式チェック

関数の呼び出し部分や定義部分で、プロトタイプ宣言に違反した記述があると、コンパイル時にエラーになります。

プロトタイプ宣言

```
int addnum(int, int);
```

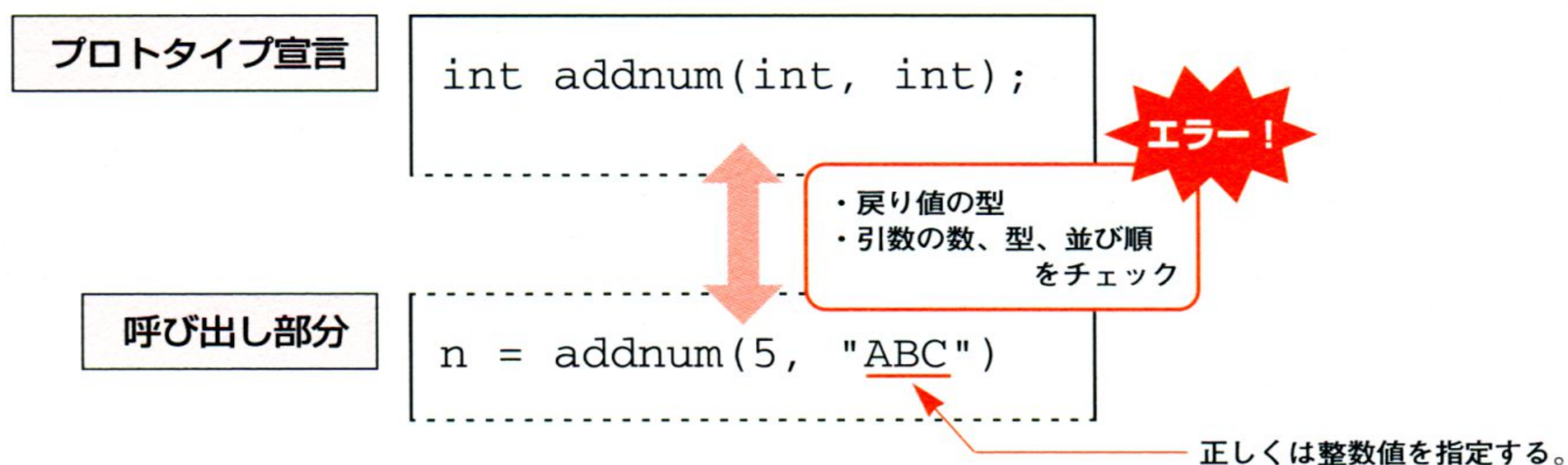
呼び出し部分

```
n = addnum(5, "ABC")
```

エラー！

・戻り値の型
・引数の数、型、並び順
をチェック

正しくは整数値を指定する。

例

```
#include <stdio.h>
void dispnum(int a);

main()
{
    int x = 10;

    dispnum(5);
    dispnum(x);
}

void dispnum(int a)
{
    printf("引数の値は：%d¥n", a);
}
```

← プロトタイプ宣言

92ページの最初の例の関数の順番を入れ替えると、このようになります。

実行結果

```
引数の値は：5
引数の値は：10
```

引数の受け渡し

値渡しと参照渡しの違いを理解しましょう。

実引数と仮引数

関数を用いる際、呼び出し側と定義側の両方で引数を指定します。C言語ではこの2つを区別しており、呼び出し側を**実引数**、定義側を**仮引数**といいます。

呼び出し部分

```
swapvalue(a, b);
```

実引数

関数の定義

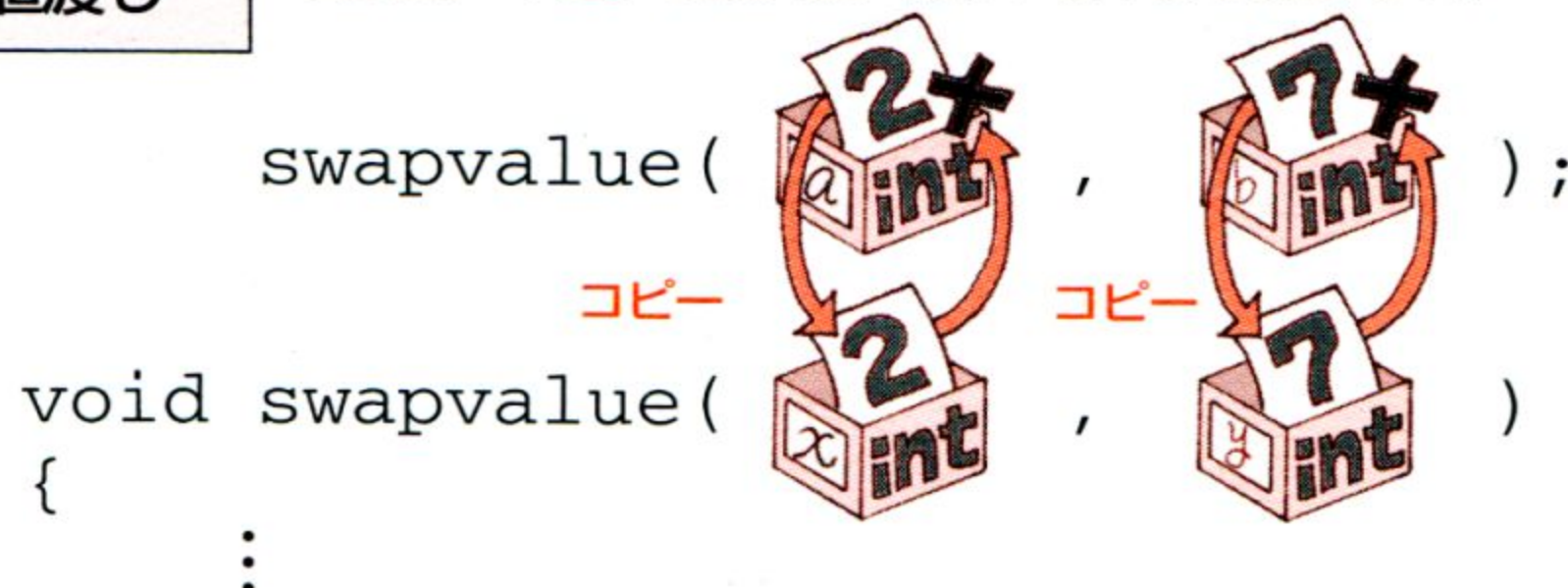
```
void swapvalue(int x, int y)
{
      ⋮
```

仮引数

値渡しと参照渡し

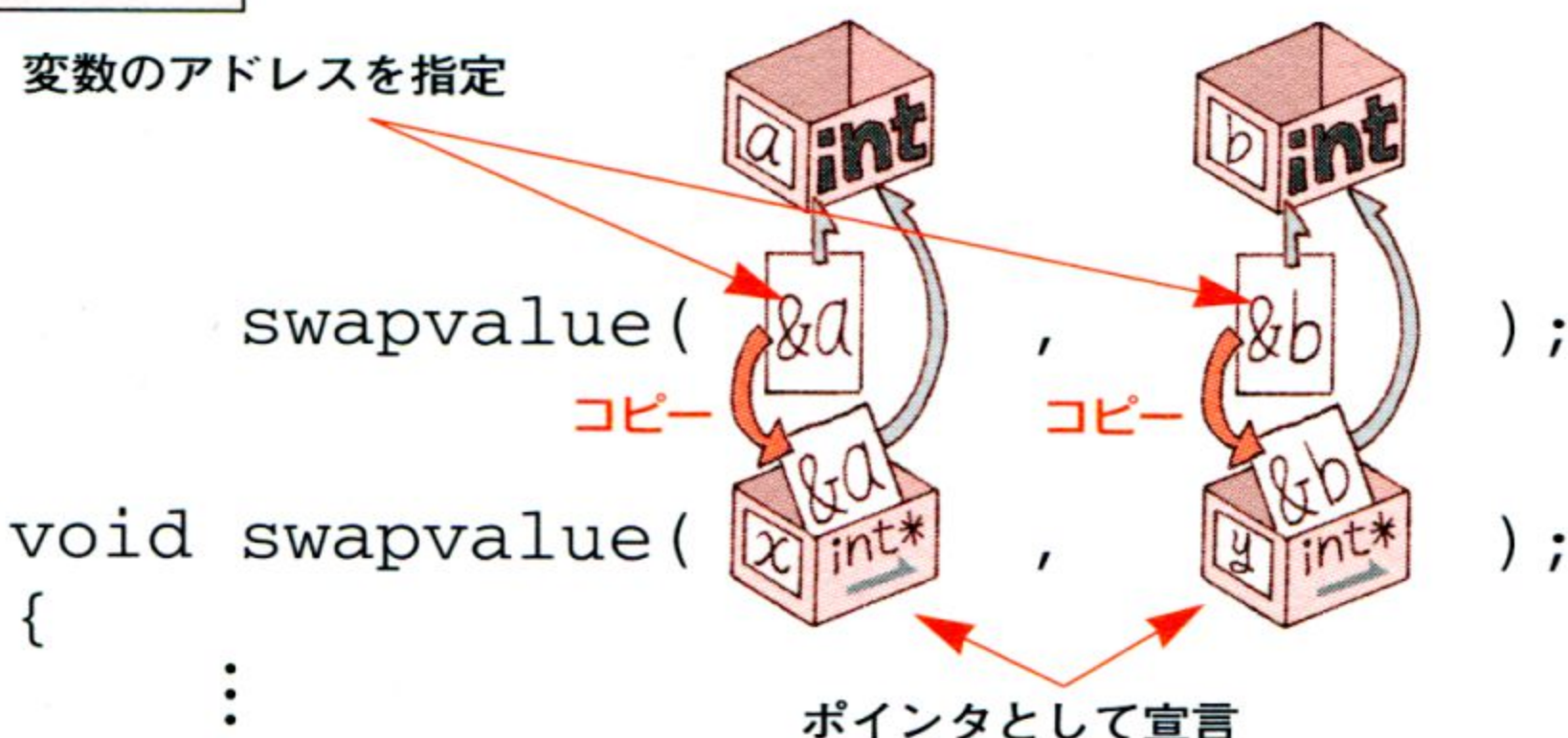
実引数と仮引数の値の受け渡し方法には、**値渡し**と**参照渡し**の2種類があります。

値渡し 実引数の「値」を仮引数に渡す、標準的な方法です。

```
     swapvalue(  ,  );
void swapvalue(  ,  )
{
     ⋮
```


実引数と仮引数は全く別の変数であると考えるため、関数の中で仮引数の値を変更しても、実引数の値には影響しません。

参照渡し 実引数の「アドレス」を、仮引数に渡す方法です。

```
     swapvalue(  ,  );
void swapvalue(  ,  );
{
     ⋮
```


実引数も仮引数も、同じアドレスの値を参照することになるので、関数の中から、呼び出し側の値を変更できます。

関数から複数の値や文字列を返すときに使います。

例

```
#include <stdio.h>

void swapbyval(int, int);
void swapbyref(int *,int *);

main()
{
    int a = 2, b = 7;

    printf("a=%d、b=%d¥n", a, b);
    swapbyval(a, b);
    printf("a=%d、b=%d¥n", a, b);
    swapbyref(&a, &b);
    printf("a=%d、b=%d¥n", a, b);
}

void swapbyval(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

void swapbyref(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

値渡し

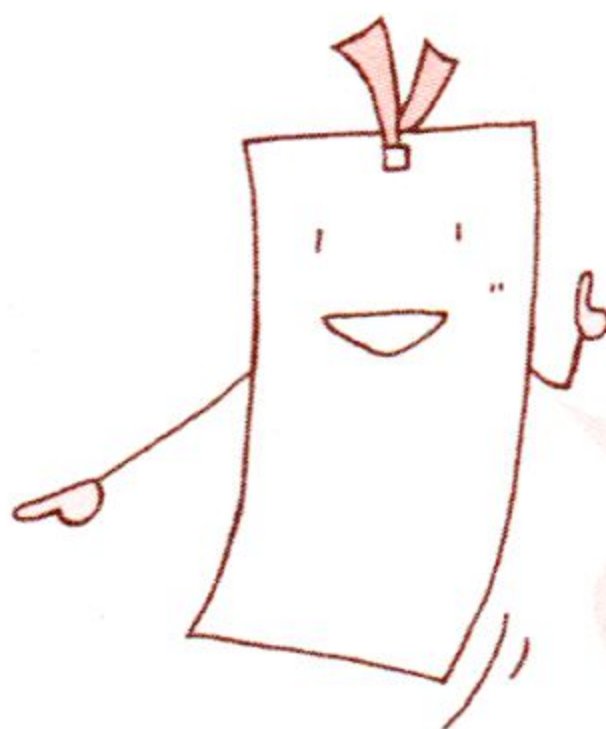
参照渡し

xとyの値を入れ替える処理

ポインタとして宣言

*xと*yの値を入れ替える処理

実行結果

```
a=2、b=7
a=2、b=7
a=7、b=2
```

参照渡しで引数を渡したときだけ、実引数を変更することができました。

main( )関数

コマンドライン引数の使い方を中心に、main()関数を理解しましょう。

main()関数の書式

main()関数は、プログラムの開始地点（エントリポイント）となる特別な関数です。これまではmain()関数を最小限の書式で記述してきましたが、次のように関数の戻り値や引数を指定する場合もあります。

```
main()
{

}
```

引数と戻り値を省略

```
void main()
{

}
```

引数を省略、戻り値はvoid

```
int main()
{
    return 0;
}
```

引数を省略、戻り値はint

```
int main(int argc, char *argv[])
{
    return 0;
}
```

正常動作したときは普通0を返します

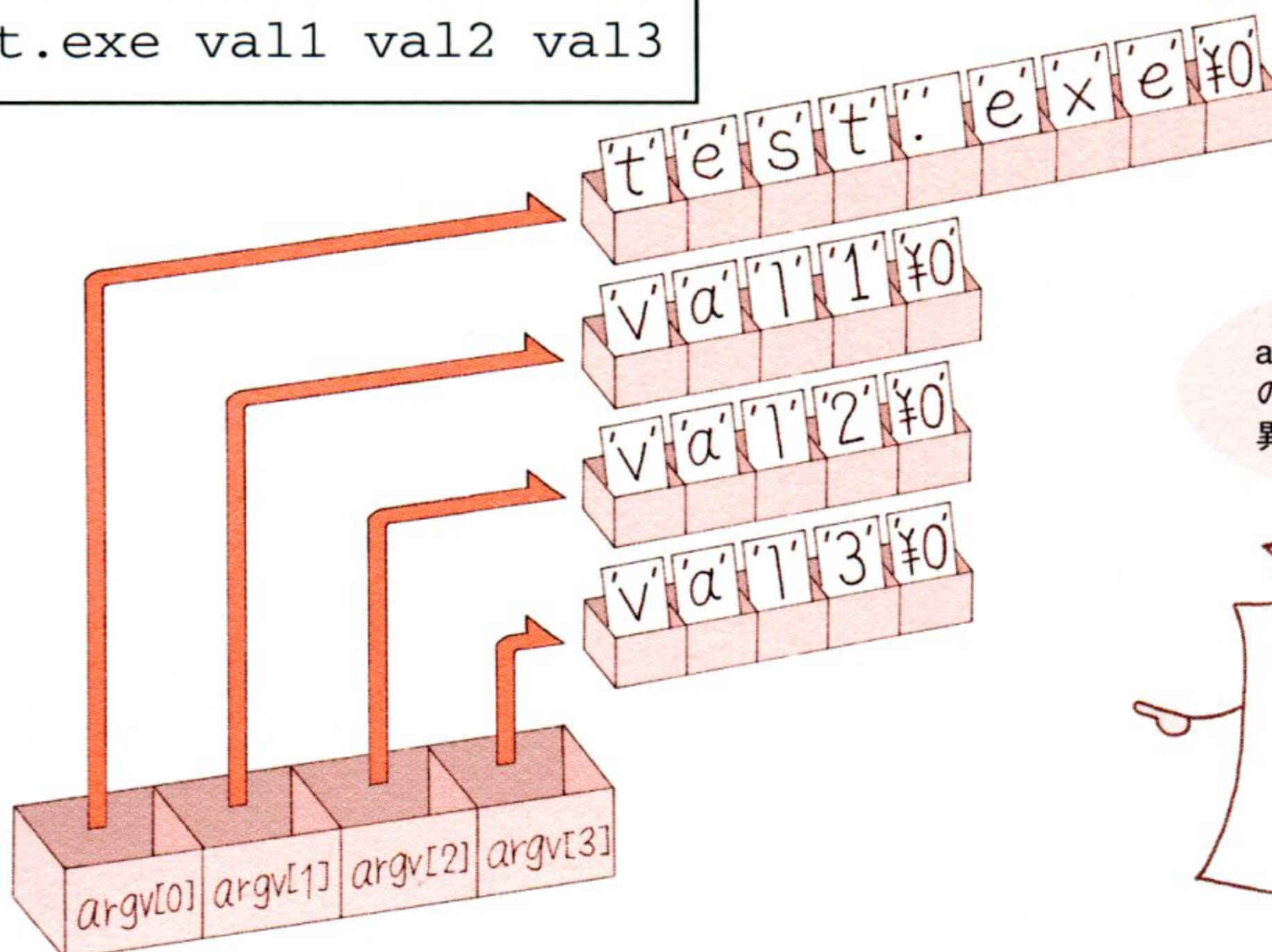
引数と戻り値（int）を指定（基本パターン）

コマンドライン引数の取得

コマンドラインから引数をつけてプログラムを実行すると、main()関数の引数に、プログラム自身のファイル名とコマンドライン引数の情報が入ります。

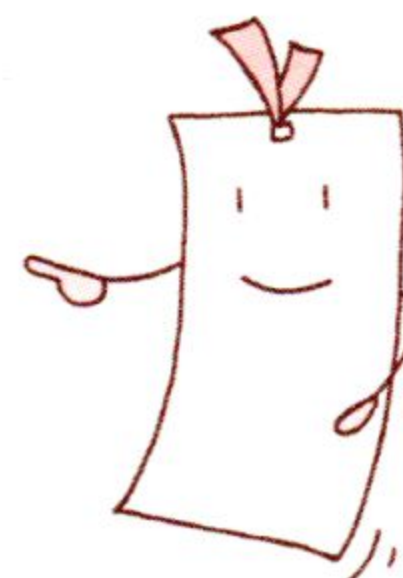
引数	格納する情報
argc	配列argvの大きさ（＝コマンドライン引数の数＋1）
argv[0]	プログラムファイルのパスの文字列へのポインタ
argv[1]	1番目のコマンドライン引数の文字列へのポインタ
argv[2]	2番目のコマンドライン引数の文字列へのポインタ
⋮	⋮

argvはポインタ配列になっています。

```
> test.exe val1 val2 val3
```

argv [0]が指す文字列の内容は処理系により異なります。

例

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int i;

    if(argc <= 1)
        return 1;
    if(strcmp(argv[1], "enum") == 0)
        for(i = 0;i<argc;i++)
            printf("argv[%d] : %s¥n", i, argv[i]);
    else if(strcmp(argv[1], "count") == 0)
        printf("コマンドライン引数の数：%d¥n", argc-1);
    return 0;
}
```

引数を指定しなかったときにエラーにならないようにします。

左記をコンパイルしたファイル名：cmdparam.exe

実行結果

```
> cmdparam.exe enum orange apple
argv[0] : cmdparam.exe
argv[1] : enum
argv[2] : orange
argv[3] : apple
> cmdparam.exe count orange apple
コマンドライン引数の数：3
```

※太字はキーボードから入力した文字

サンプルプログラム

■ファイルを削除する

コマンドライン上でファイルを削除するコマンド（delやrm）を実行すると、ファイルを復元できません。そこで、引数で指定したファイル名に.bakを付加し、退避するプログラム"trash"を作ります。ただし、-dオプションを付加すると、ファイルを完全に削除するものとします。

ソースコード

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
    char usage[] = "usage: trash <-d> filename¥n";
    int ret = 0; /* 関数の戻り値 */
    char newfilename[256] = "";

    /* パラメータなし */
    if(argc <= 1) {
        printf(usage);
        return 1;
    }
    /* -d指定あり */
    else if(strcmp(argv[1], "-d") == 0) {
        if(argc <= 2) {
            printf(usage);
            return 2;
        }
        ret = remove(argv[2]);
        if(ret == 0)
            printf("ファイルを削除しました。¥n");
        else
            printf("ファイルを削除できませんでした。¥n");
    }
    /* -d指定なし */
    else {
        sprintf(newfilename, "%s.bak", argv[1]);
        ret = rename(argv[1], newfilename);
        if(ret == 0)
            printf("ファイル名の最後に.bakを付加しました。¥n");
        else
            printf("ファイル名の変更ができませんでした。¥n");
    }
}
```

remove()関数（リムーブ）
引数で指定したファイルを削除します。

rename()関数（リネーム）
第1引数で指定したファイルの名前を第2引数の名前に変更します。

実行結果

a.txtとb.txtがカレントディレクトリに存在するとします。

```
>trash a.txt ↵
ファイルの最後に.bakを付加しました。
>trash -d b.txt ↵
ファイルを削除しました。
```

※太字はキーボードから入力した文字

a.txt.bakができていて、b.txtが削除されていることを確認してください。

■西暦から和暦を求める

1990などの西暦の値を入力すると、平成2年という和暦を出力するプログラムを作ります。

ソースコード

```
#include <stdio.h>
#include <string.h>
int wtoj(int, char *, int *);

int main()
{
    int wyear = -1, jyear = 0;
    char nengo[16];

    printf("西暦→和暦の変換を行います。終了するには0を入力してください。¥n");
    while(wyear != 0) {
        printf("西暦を入力してください(1868-2050) : ");
        scanf("%d", &wyear);
        if(wtoj(wyear, nengo, &jyear) == 0)
            printf("西暦%d年は、%s%d年です¥n", wyear, nengo, jyear);
    }
    return 0;
}

/*********************************************
  wtoj() 西暦から和暦へ変換する
  [引数] wyear -- 西暦
         nengo -- 和暦の年号文字列へのポインタ
         jyear -- 和暦へのポインタ
  [戻り値] 変換できたら0、範囲外なら1
*********************************************/
int wtoj(int wyear, char *nengo, int *jyear)
{
    if(wyear >= 1868 && wyear <= 1911) {
        strcpy(nengo, "明治");
        *jyear = wyear-1868+1;
        return 0;
    } else if(wyear >= 1912 && wyear <= 1925) {
        strcpy(nengo, "大正");
        *jyear = wyear-1912+1;
        return 0;
    } else if(wyear >= 1926 && wyear <= 1988) {
        strcpy(nengo, "昭和");
        *jyear = wyear-1926+1;
        return 0;
    } else if(wyear >= 1989 && wyear <= 2050) {
        strcpy(nengo, "平成");
        *jyear = wyear-1989+1;
        return 0;
    }
    return 1;
}
```

wtoj()関数のプロトタイプ宣言

入力した西暦を整数としてwyearに格納しています（→120ページ）。

関数の説明を書くとよいでしょう。

複数の値を取得するときは参照渡しにします。

実行結果

※太字はキーボードから入力した文字

```
西暦→和暦の変換を行います。終了するには0を入力してください。
西暦を入力してください(1868-2050) : 1990 ↵
西暦1990年は、平成2年です
西暦を入力してください(1868-2050) : 0↵
```

COLUMN コラム

～再帰呼び出し～

関数は、自分自身を呼び出すことができます。このことを、**再帰呼び出し**といいます。たとえば、次のような感じです。

```
void func(int c)
{
    printf("Hello!¥n");
    c--;
    if(c > 0)          ← 終了条件
        func(c);       ← 自分自身を呼び出す
}
```

この関数をmain()関数などから、「func(5);」と呼び出すと、"Hello!"という文字列を5回表示します。

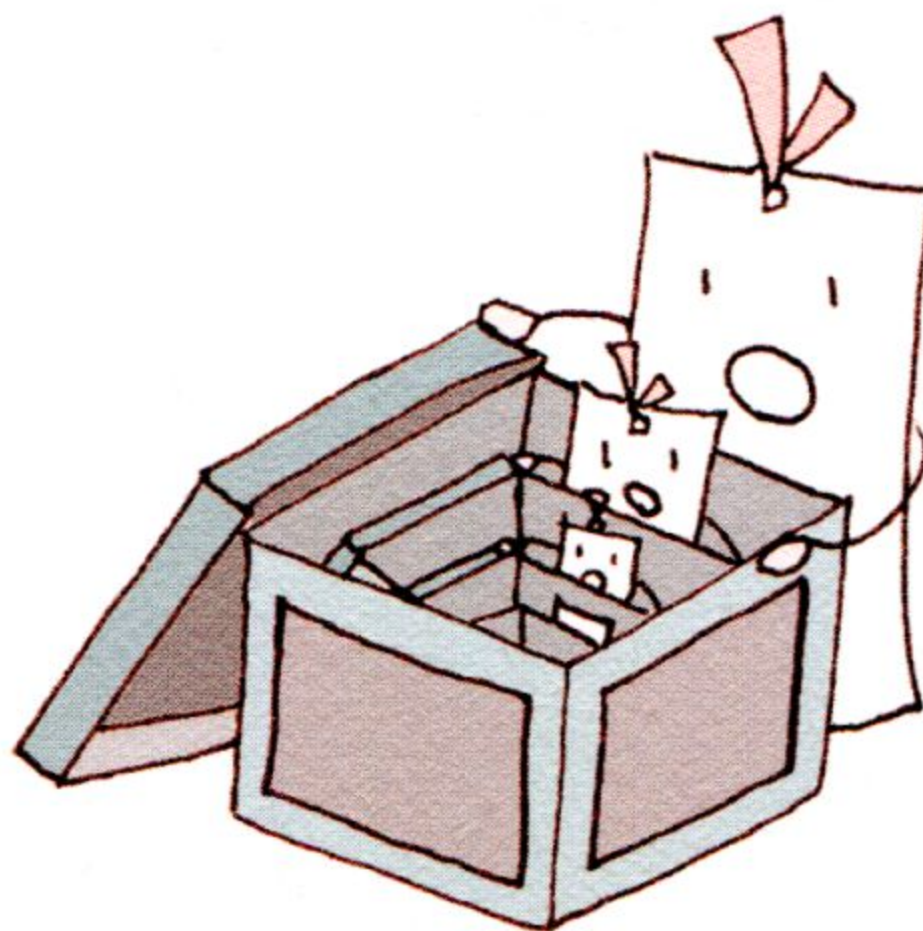
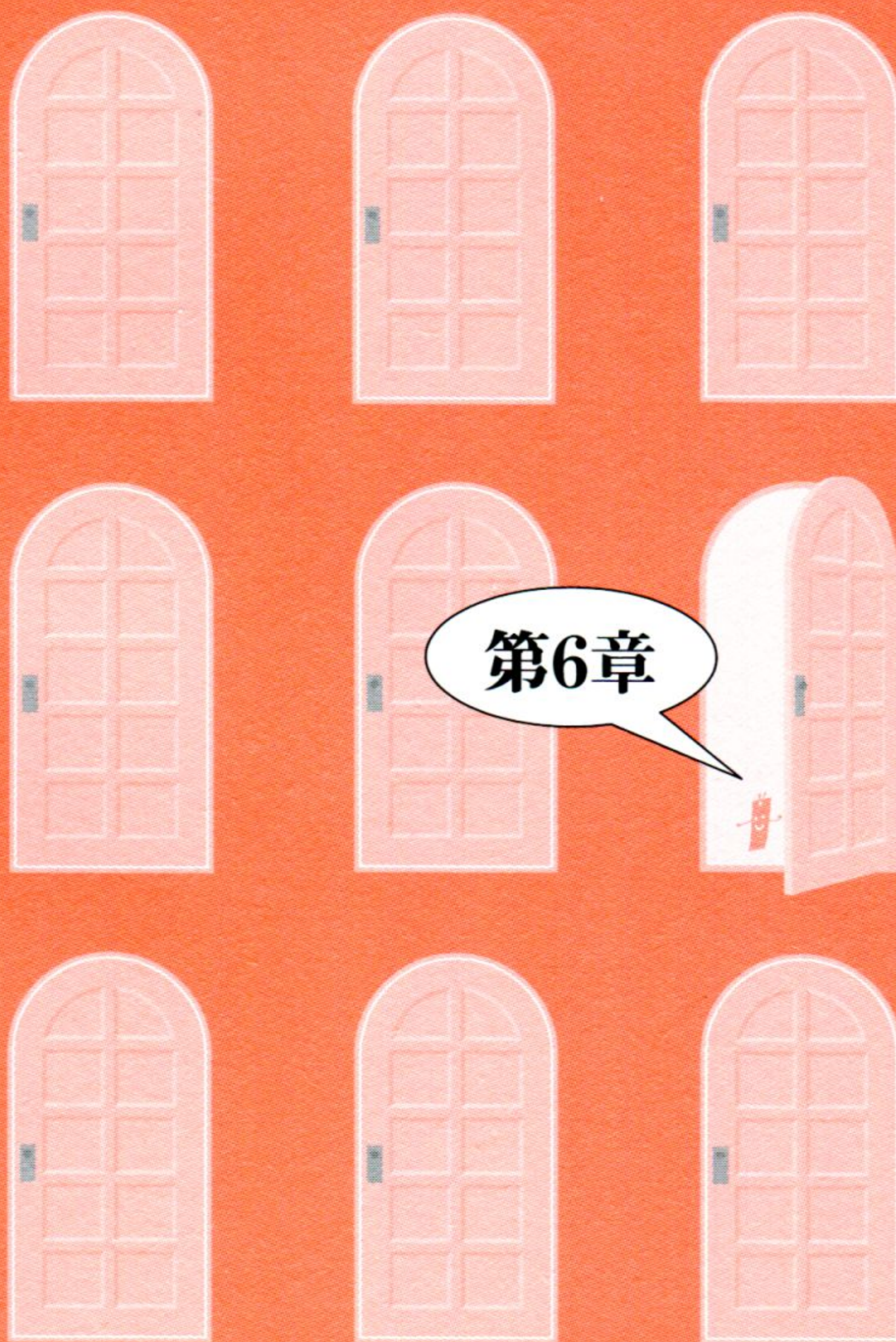
さて、再帰呼び出しの関数を作るにあたって、重要な注意点があります。それは、再帰呼び出しでは、必ず**終了条件を指定**するということです。もし、終了条件がなかったらどうなるでしょう？上の例で「if(c > 0)」の部分がないと、文字列"Hello!"を表示してfunc()関数を呼び出す…という手順を延々と繰り返し、いつまで経ってもプログラムが終了しなくなってしまいます。実際には、再帰呼び出しを無限に続けると、スタックというメモリ領域が足りなくなって、アプリケーションエラーが発生してしまいます。再帰呼び出しのときは必ず終了条件を正しく指定するようにしてください。

再帰呼び出しの利点は、なんといっても、複雑な処理をスマートに記述できることです。再帰呼び出しの典型的な例として、整数nの階乗（$n! = n \times (n-1) \times \cdots \times 2 \times 1$）を求める関数を紹介しておきましょう。

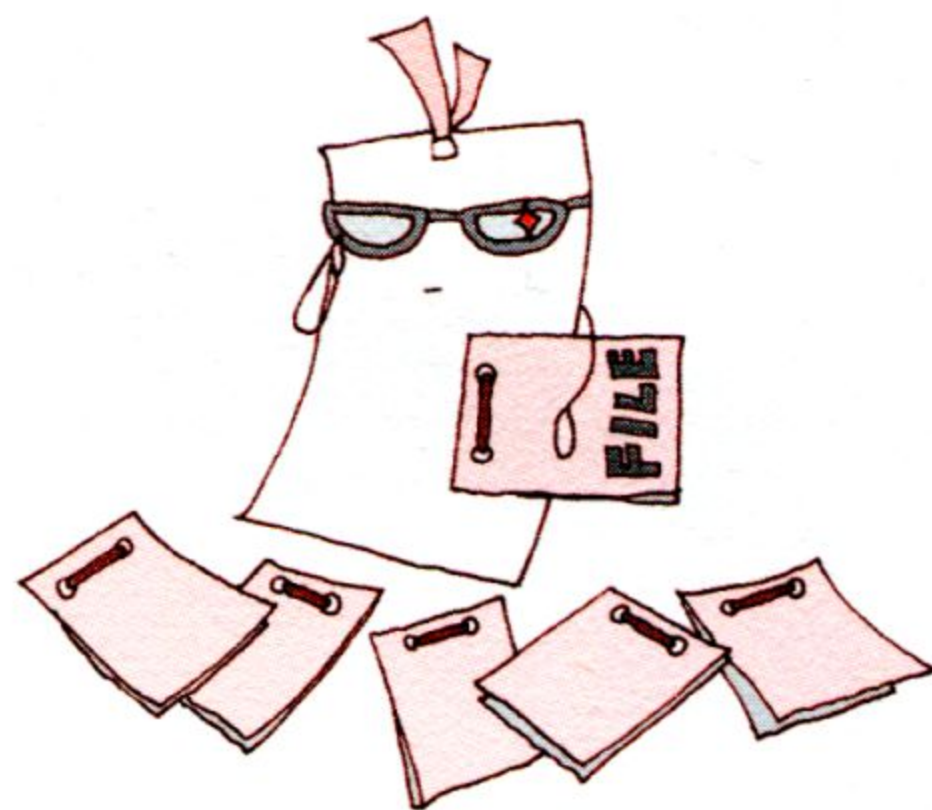
```
int kaijo(int n)
{
    if(n == 0)
        return 1;
    else
        return (n*kaijo(n-1));
}
```

6 ファイルの入出力

第6章

第6章は ここがkey!

ファイルって何だろう？

この章ではファイルについて学びます。ファイルというと、普通は「何かのソフトで作ったデータを保存したもの」をイメージするのではないでしょうか。C言語でいうファイルにはもちろんそういう意味もありますが、実はそれだけではありません。

本題に入る前に、ファイルの種類について考えてみましょう。ファイルは大きく**テキストファイル**と**バイナリファイル**に分かれます。両者の見分け方は、「人間が読めるか読めないか」です。つまり、テキストファイルの方は、人間がわかるようなルールで記録してあるファイルなのです。それに対し、バイナリファイルは、意味不明のデータの羅列にしか見えません。身近な例では、今まで作ってきたC言語のソースプログラムはテキストファイル、それをコンパイルしたものはバイナリファイルになります。

ファイルを扱うには手順がある

それをふまえた上で、プログラム上でファイルの中身を読んだり、ファイルにデータを書き加えたりする方法を見ていきます。C言語をはじめ、多くのプログラミング言語では、ファイル名を直接指定して読み書きしたりはしません。その代わり、**ファイルポインタ**というものでファイルを置き換え、これを通してファイルにアクセスします。このファイルポインタはファイルのどの部分に対して読み書きを行うかという情報も含んでいます。

ファイルポインタを宣言したら、「①対象のファイルを開き、ファイルポインタを得る　②ファイルポインタを通して読み書きする　③操作がおわったら、ファイルを閉じる」という3ステップでプログラムを作っていきます。テキストファイルとバイナリファイルでは関数の種類や引数が微妙に違うため、混乱しないようにしましょう。

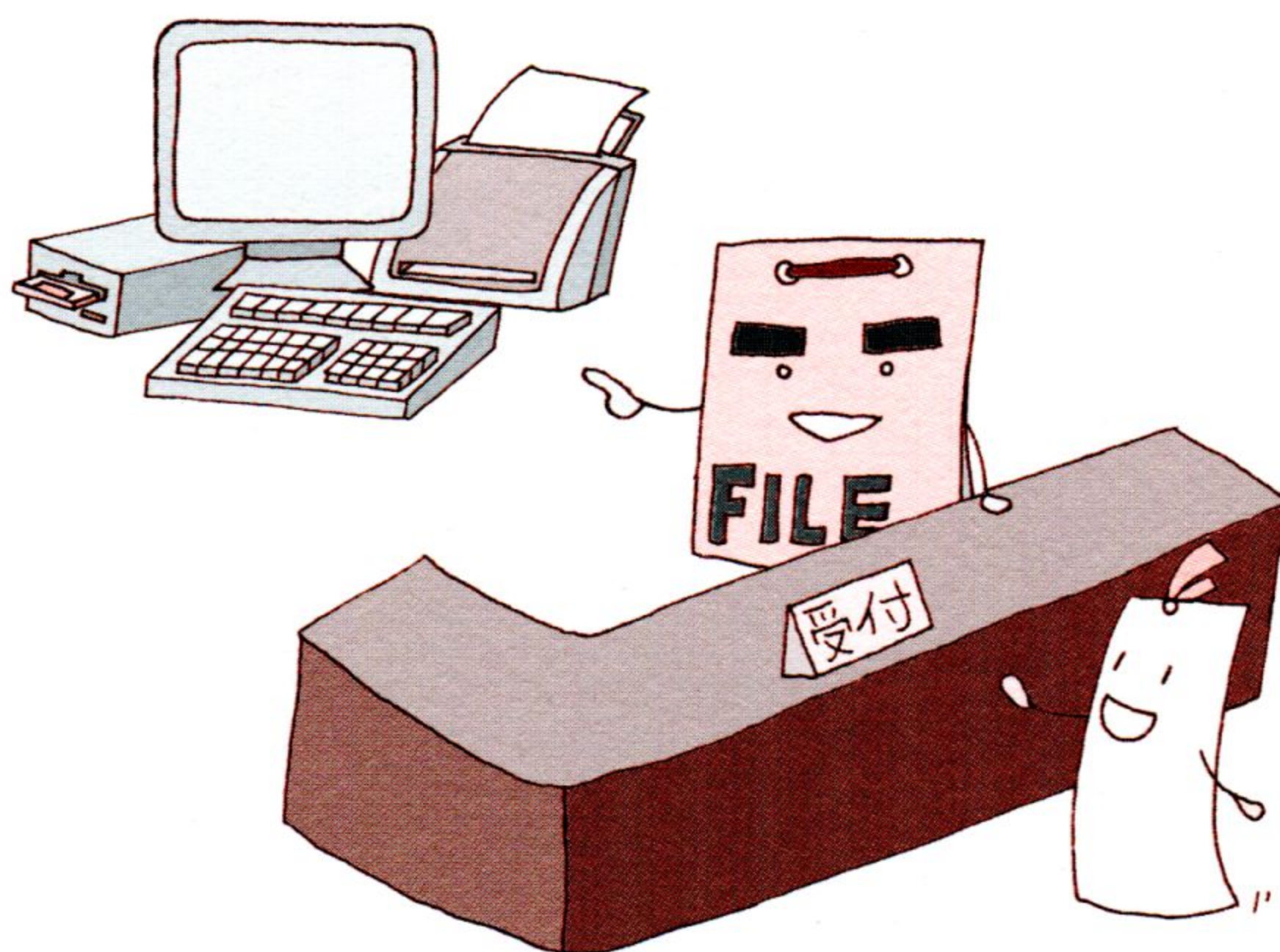
Topics

キー入力でもファイルを使う

ここまでは、普通のディスク上に存在するファイルのお話でしたが、C言語のファイルはもう少し広い意味があります。

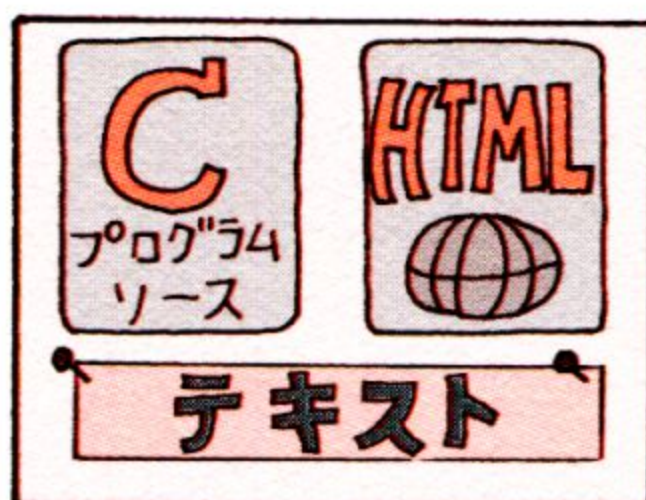
たとえば、キーボードから文字を入力したデータを読み込み、ディスプレイに表示するというプログラムがあるとしましょう。このプログラムには、キーボードからプログラムへデータを読み込む**入力部分**と、プログラム内のデータをディスプレイに表示する**出力部分**があることになります。これら入出力全般は、対象がディスク上のファイルであろうと、キーボードからの入力であろうと、「ファイル」という考え方を使って共通化できるのです。

キーボードやディスプレイを扱うためのファイルのことを**標準入出力ファイル**といいます。標準入出力ファイルは、データファイルのように具体的なイメージには結びつきませんが、プログラムの実行開始と同時に開いており、いつでも使える状態になっています。縁の下の力持ち、という感じでしょうか？

データを変数に格納しただけではプログラムが終了すればデータは消えてしまいますが、ハードディスクなどのファイルに保存しておけば、電源を落としても消えずに残ります。ファイルの使い方をマスターし、データをうまく活用できるようになればきっとプログラムの幅が広がります。

ファイル

データやプログラムなどをディスク上に記録したものをファイルといいますが、どのように扱っていけばいいのでしょう？

ファイルの種類

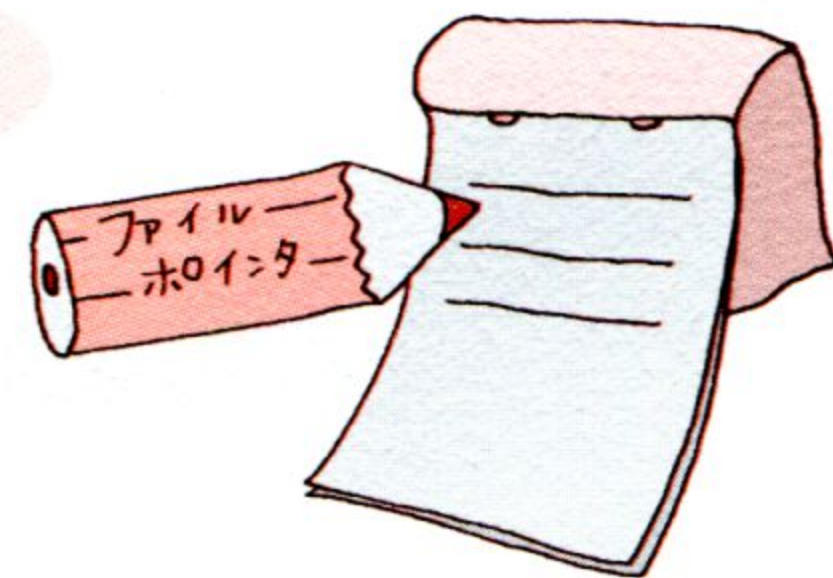
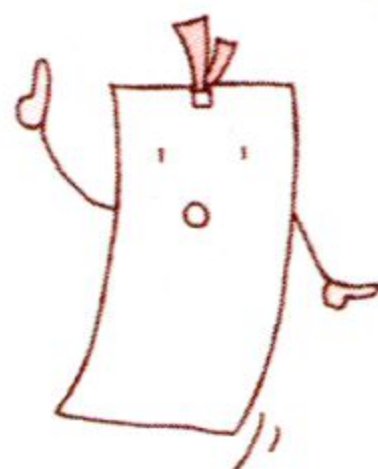
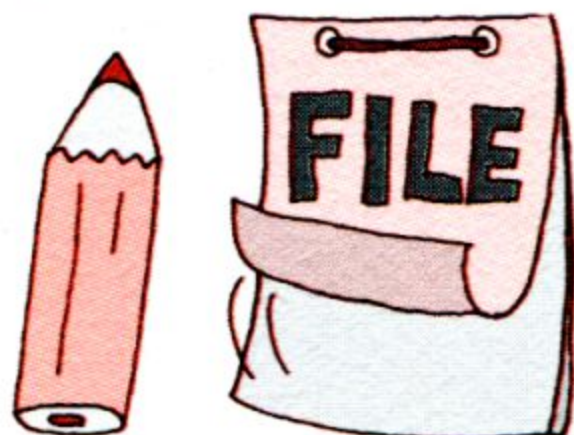
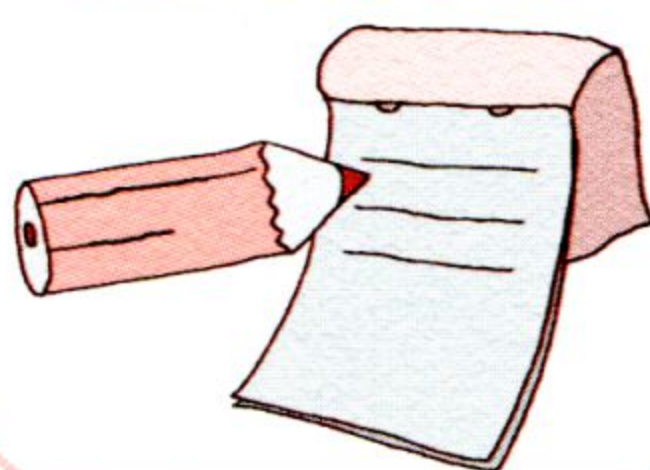
ファイルには、大きく分けてテキストファイルとバイナリファイルの2種類があります。バイナリファイルはテキストエディタでは文字として読めません。

文字として読めるもの
（C言語のプログラムソースやHTMLなど）

文字として読めないもの
（コンパイル後のC言語プログラム、画像データなど）

ファイル処理の基本

プログラムでファイルを扱うにはあらかじめファイルポインタの宣言が必要になります。ファイルポインタはファイルの読み書きをはじめる位置を示す目印のようなものです。宣言は次のように行います。

```
FILE *fp;
```

ファイルポインタ

そして、ファイルを扱うときは必ず次の手順でプログラムを記述していきます。

①ファイルを開く → ②読み書きを行う → ③ファイルを閉じる

»ファイルを開く

ファイルを開くには**fopen()**関数を使います。

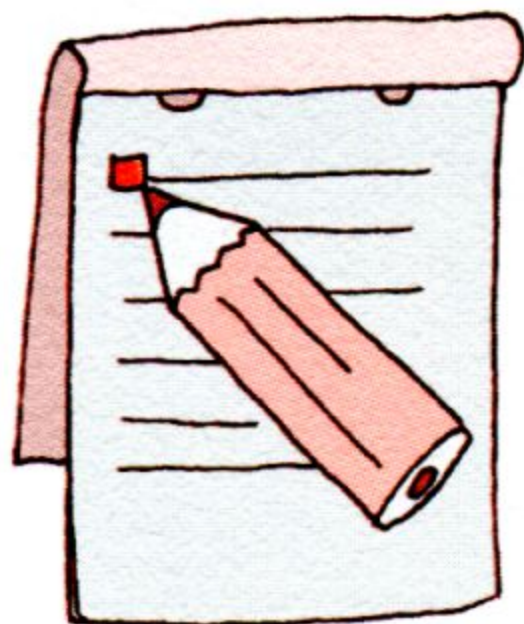
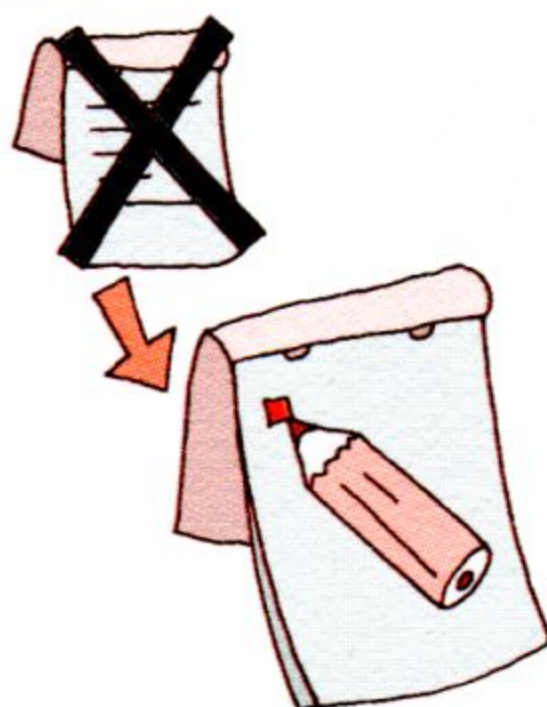
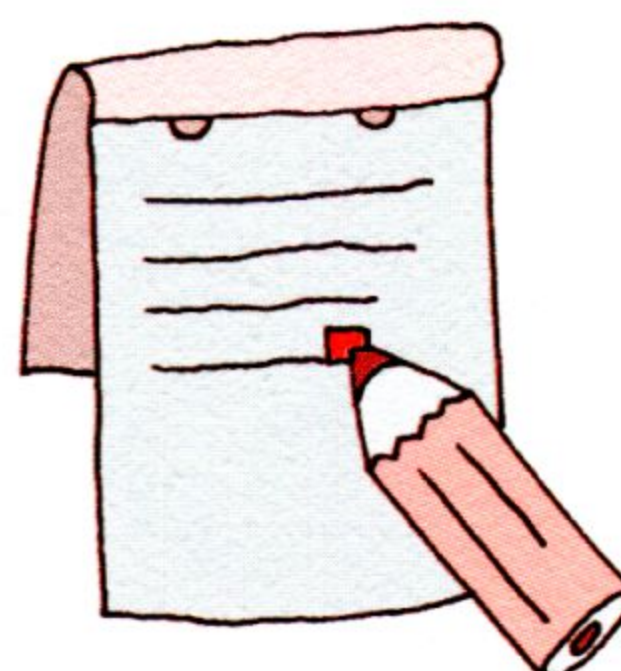
ファイル名でパスを指定しないと、カレントディレクトリを参照します。

```
FILE *fp;
fp = fopen("file1.txt", "r");
```

ファイルポインタ　ファイル名　オープンモード

ファイルを開く方法を指定します。
主なオープンモードは次のとおりです。

```
"r" ……読み出し専用
"w" ……書き込み専用
"a" ……追加書き込み
```

オープンに成功すると、fopen()関数はファイルポインタを返します。どのモードでファイルを開くかによって、前のファイルの扱いや、ファイルポインタが最初に示す位置は異なります。

"r"
ファイルの先頭

"w"
ファイルの先頭
（すでにあるファイルはなくなります）

"a"
ファイルの末尾
（ファイルがない場合は新規に作ります）

読み込むファイルがない、書き込み権限がない、などの理由でオープンに失敗すると、fopen()関数はNULLを返します。ファイルポインタがNULLでないかのチェックは必ず行いましょう。

»ファイルを閉じる

ファイルを閉じるには（オープンモードにかかわらず）**fclose()**関数を使います。

```
fclose(fp);
```

ファイルの読み込み

テキストファイルの読み込みを例にファイルを扱う流れを見ていきます。

テキストファイルの読み込み手順

file1.txtから1行分のデータを文字配列sに読み込んでみましょう。

①ファイルを開く

オープンモードを読み出し専用の"r"にしてファイルを開きます。

```
char s[10];
FILE *fp;
fp = fopen("file1.txt", "r");
```

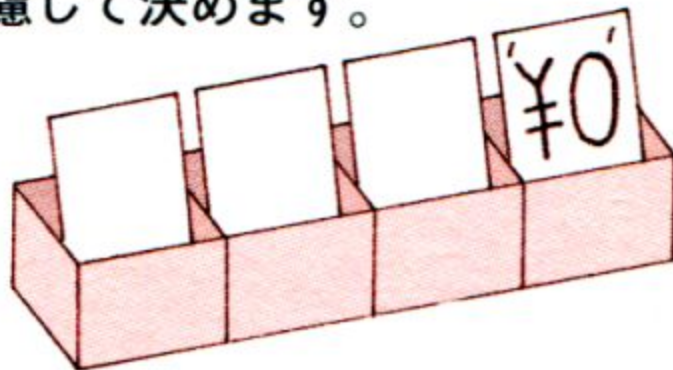
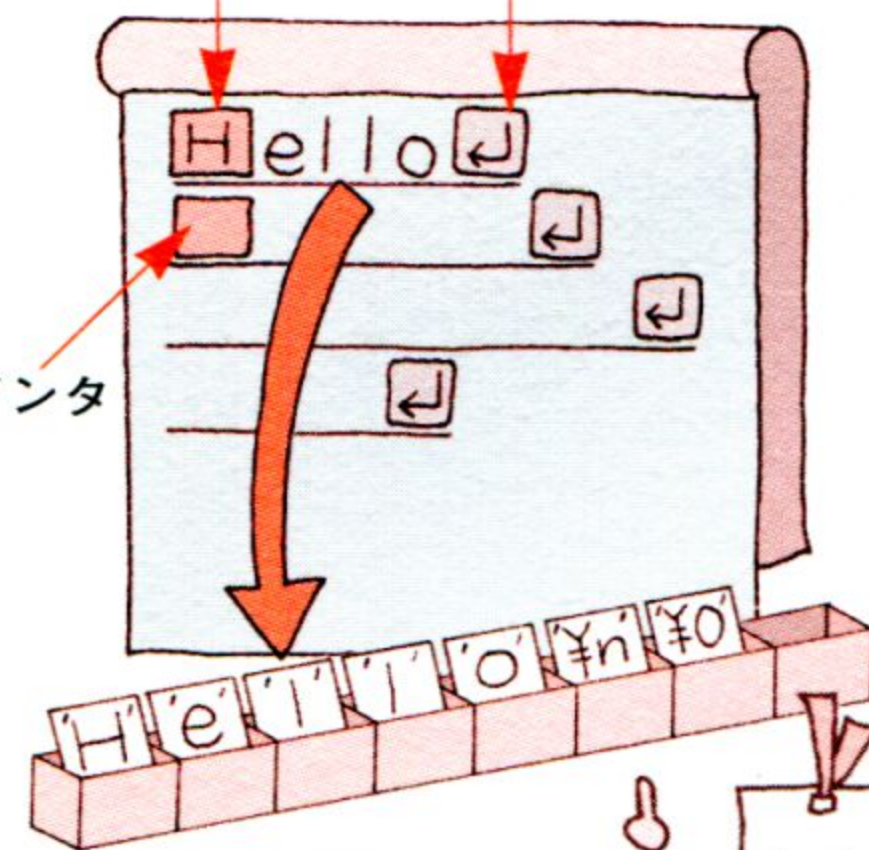
読み込んだデータの格納用文字列

②データを読み込む

1行分読み込むときは、**fgets()**（エフゲットエス）関数を使います。
次のようにすると、fpが示す位置から最大9文字をsに格納します。

```
fgets(s, 10, fp);
```

格納用文字配列
読み込み最大文字数
ファイルポインタ

配列の大きさと最大文字数は、自動的に付くNULL文字分を考慮して決めます。

ファイルポインタの初期位置
改行コードまでか、最大文字数分を読み込む。
ファイルポインタの次の位置
読み込んだ文字列の最後には改行がつきます。

③ファイルを閉じる

最後にfclose()関数を使ってファイルを閉じます。

```
fclose(fp);
```

≫ファイルの最後まで読む

ファイルを1行ずつ最後まで読み込むには、fgets()をファイルのおわりが来るまで繰り返します。

ファイルの終わりを調べるには**feof()**（エフイーオーエフ）関数を使います。feof()関数はファイルの終端（**e**nd **o**f **f**ile）のときに真（true）になる関数です。

```
        :
while(1)
{
    fgets(s, 10, fp);
    if(feof(fp))
        break;
}
        :
```

ファイルの終端に来るとfeof( )関数はtrueになり、ループが終了します。

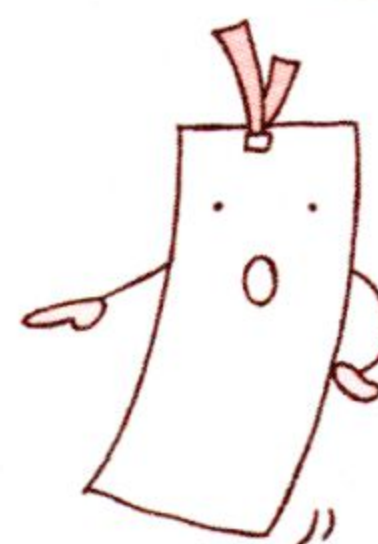
あらかじめabc.txtを用意しておきます。

例

```
#include <stdio.h>

main()
{
    FILE *fp;
    char s[20];
    int i = 1;
    fp = fopen("abc.txt", "r");
    if(fp == NULL)
        return;
    while(1)
    {
        fgets(s, 20, fp);
        if(feof(fp))
            break;
        printf("%04d:%s", i, s);
        i++;
    }
    fclose(fp);
}
```

ファイルがオープンできなかったときはプログラムを終了させます。

abc.txtの内容

```
abcdefg
hijklmn
opqrstu
vwxyz
```

実行結果

```
0001:abcdefg
0002:hijklmn
0003:opqrstu
0004:vwxyz
```

ファイルの内容を行番号つきで表示します。

ファイルの書き出し

読み込みの次は、プログラムでテキストファイルを作ってみましょう。

テキストファイルの書き出し手順

"Hello"というデータをfile2.txtに書き出してみましょう。

①ファイルを開く

オープンモードを新規書き込み用の"w"か、追加書き込み用の"a"にしてファイルを開きます。

```
FILE *fp;
fp = fopen("file2.txt", "w");
```

②ファイルに書き出す

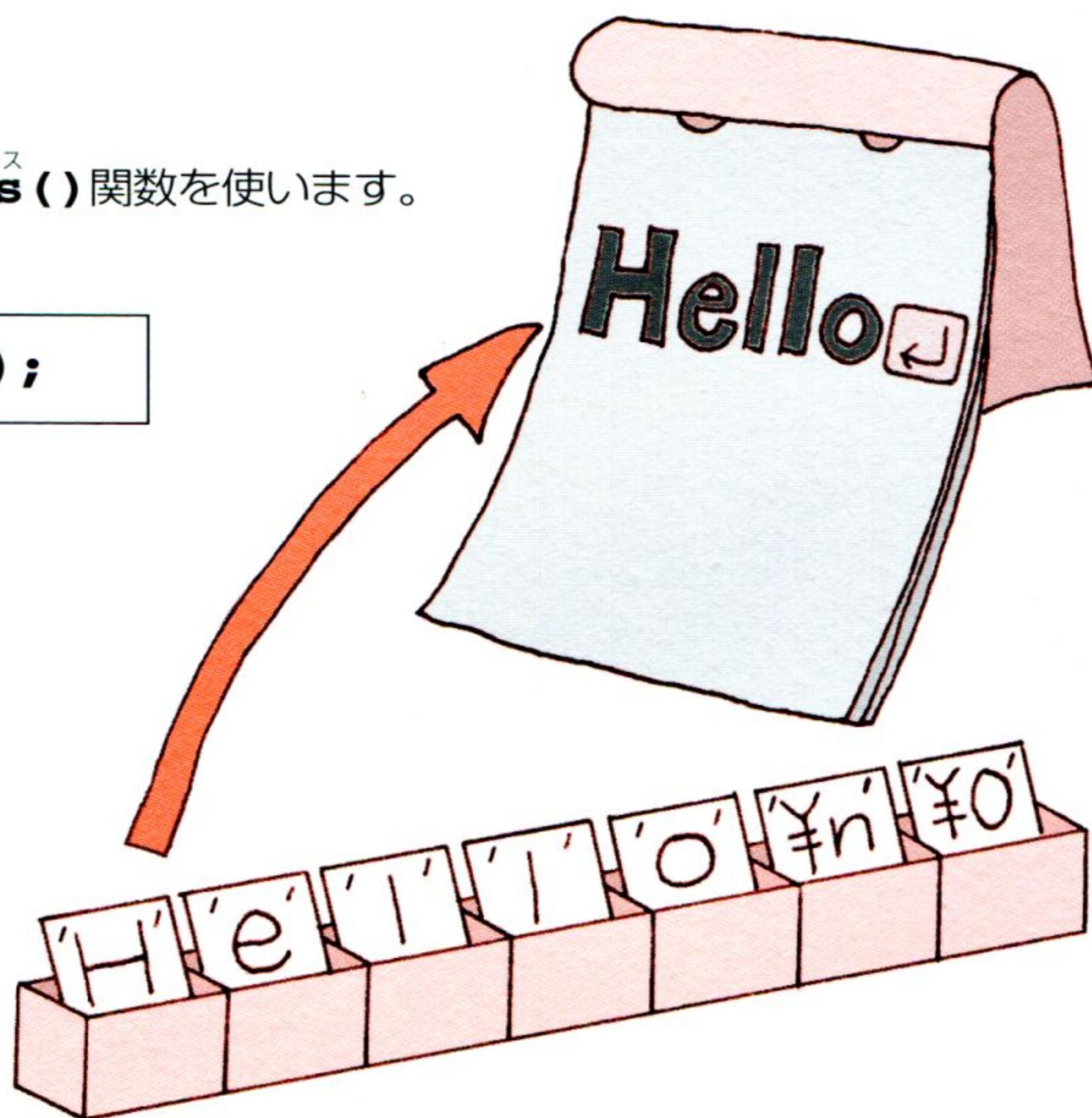
文字列を書き出すときは、fputs()（エフプットエス）関数を使います。

```
fputs("Hello¥n", fp);
```

書き出す文字列

Hello
H e l l o ¥n ¥0

③ファイルを閉じる

```
fclose(fp);
```

»指定形式での書き出し

fprintf()関数はprintf()関数と同じ働きをファイルに対して行います。テキストデータをファイルに書き出すときは、fprintf()関数の方が便利です。

```
int a = 5;
fprintf(fp, "%02d¥n", a);
```

書式指定　書き出すデータ

最初のパラメータがファイルポインタになっています。

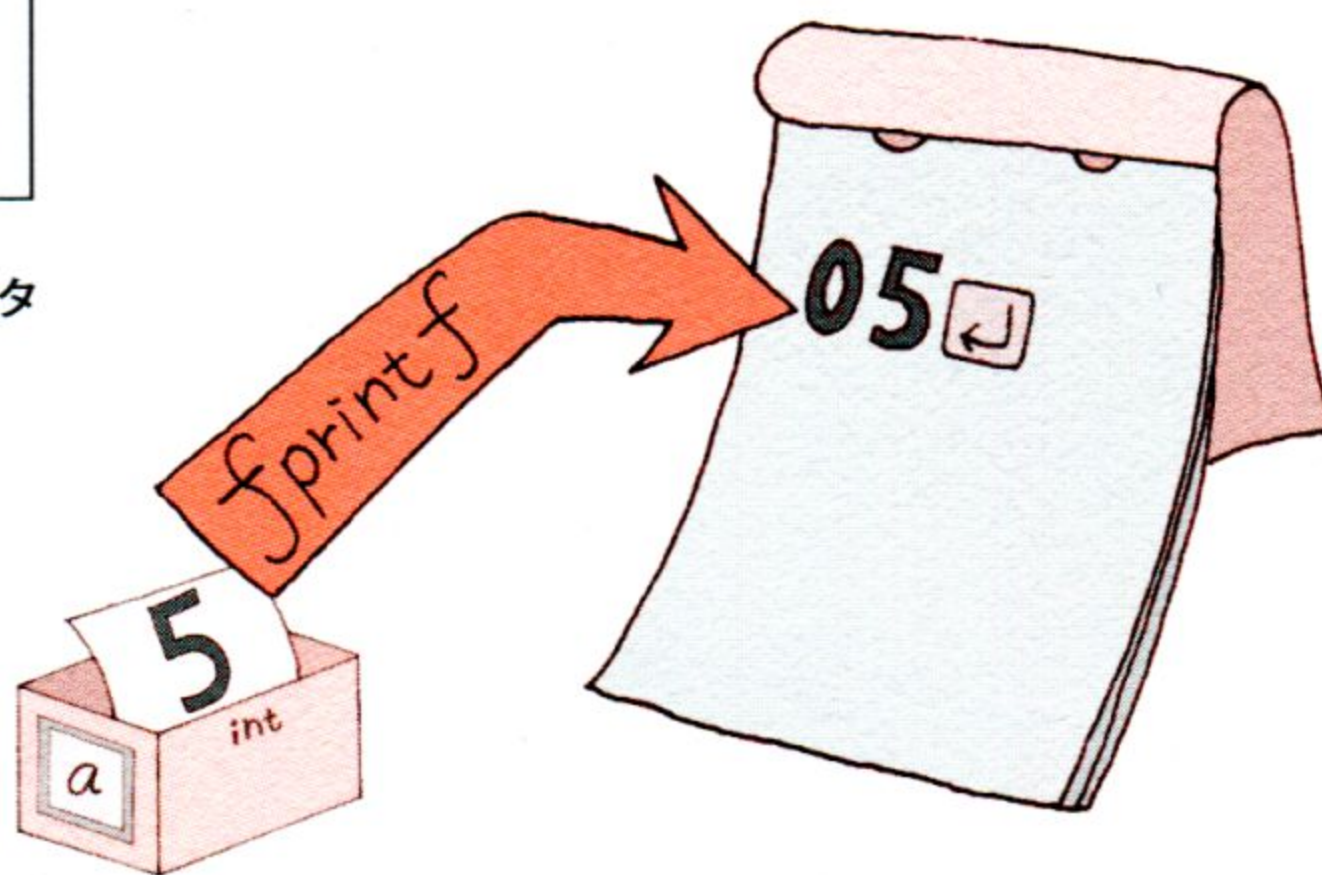
fprintf

05

5

int

a

例

```
#include <stdio.h>

main()
{
    FILE *fp;
    int a = 100, b = 5, c = 40;
    int x = 1, y = 10, z = 100;
    char delm[] = "-----------¥n";

    fp = fopen("mat.txt", "w");
    if(fp == NULL)
        return;
    fputs(delm, fp);
    fprintf(fp,"%4d%4d%4d¥n%4d%4d%4d¥n", a, b, c, x, y, z);
    fputs(delm, fp);
    fclose(fp);
}
```

実行結果

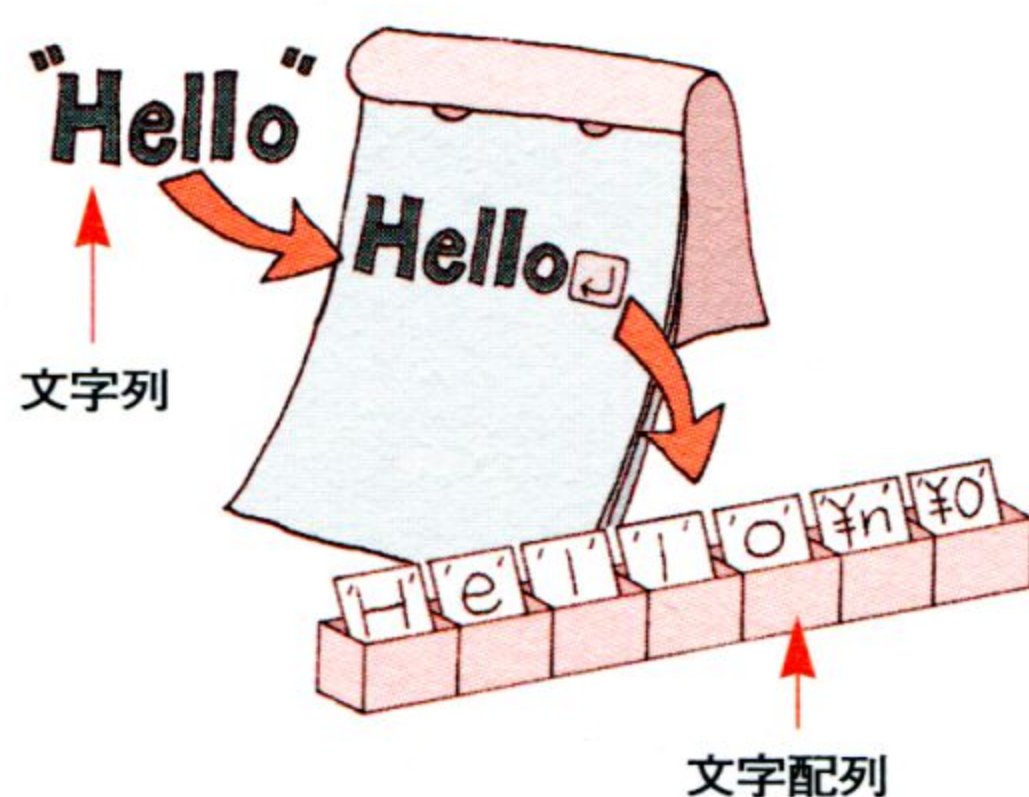
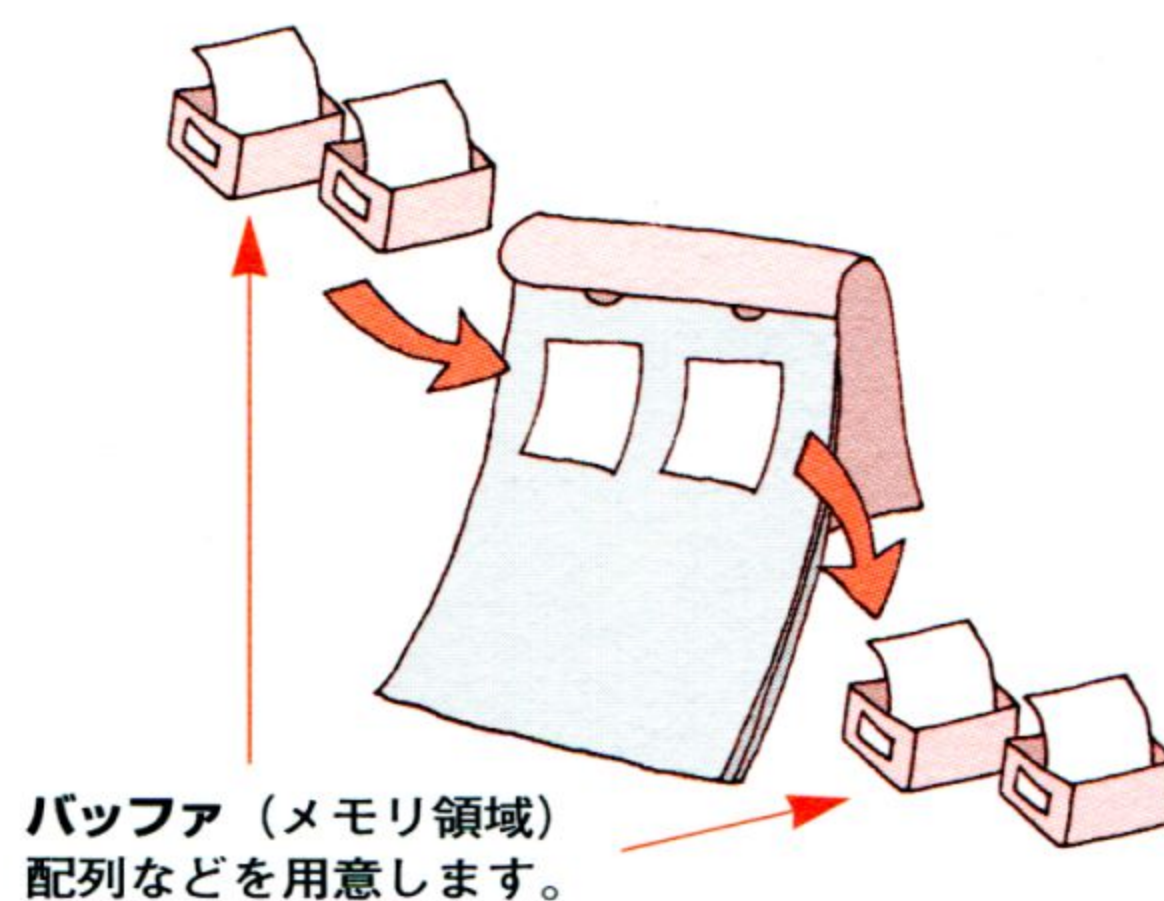
(mat.txtの中身)

```
-----------
 100   5  40
   1  10 100
-----------
```

バイナリの読み書き(1)

バイナリファイルではファイルの開き方や読み書きに使う関数がテキストファイルとは少し異なります。

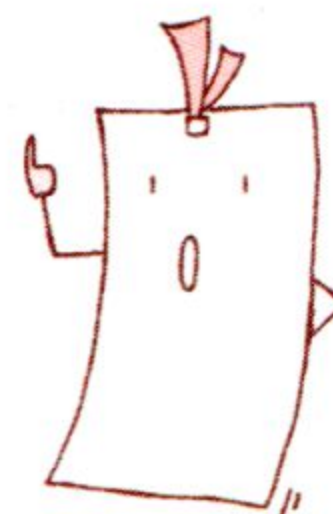
バイナリファイルの読み書き

テキストファイルの読み書きでは、fgets()などの関数が自動的に改行などを認識していました。一方バイナリファイルの場合は、通常の文字でも改行などの制御コードでも区別なく、同等のデータとして扱います。

バイナリファイルを開く

バイナリファイルを開くときもfopen()関数を使います。ただし、オープンモードで「b」を追加指定して、バイナリモードでオープンする必要があります。

```
FILE *fp;
fp = fopen("file3.dat", "rb");
```



```
"rb" ·····読み出し専用
"wb" ·····書き込み専用
"ab" ·····追加書き込み
```

バイナリファイルの読み込み手順

file3.datからshort型変数3つ分をメモリに読み込みます。

①ファイルを開く

オープンモードをバイナリ読み込み専用の"rb"にしてファイルを開きます。

```
short buf[3];
FILE *fp;
fp = fopen("file3.dat", "rb");
```

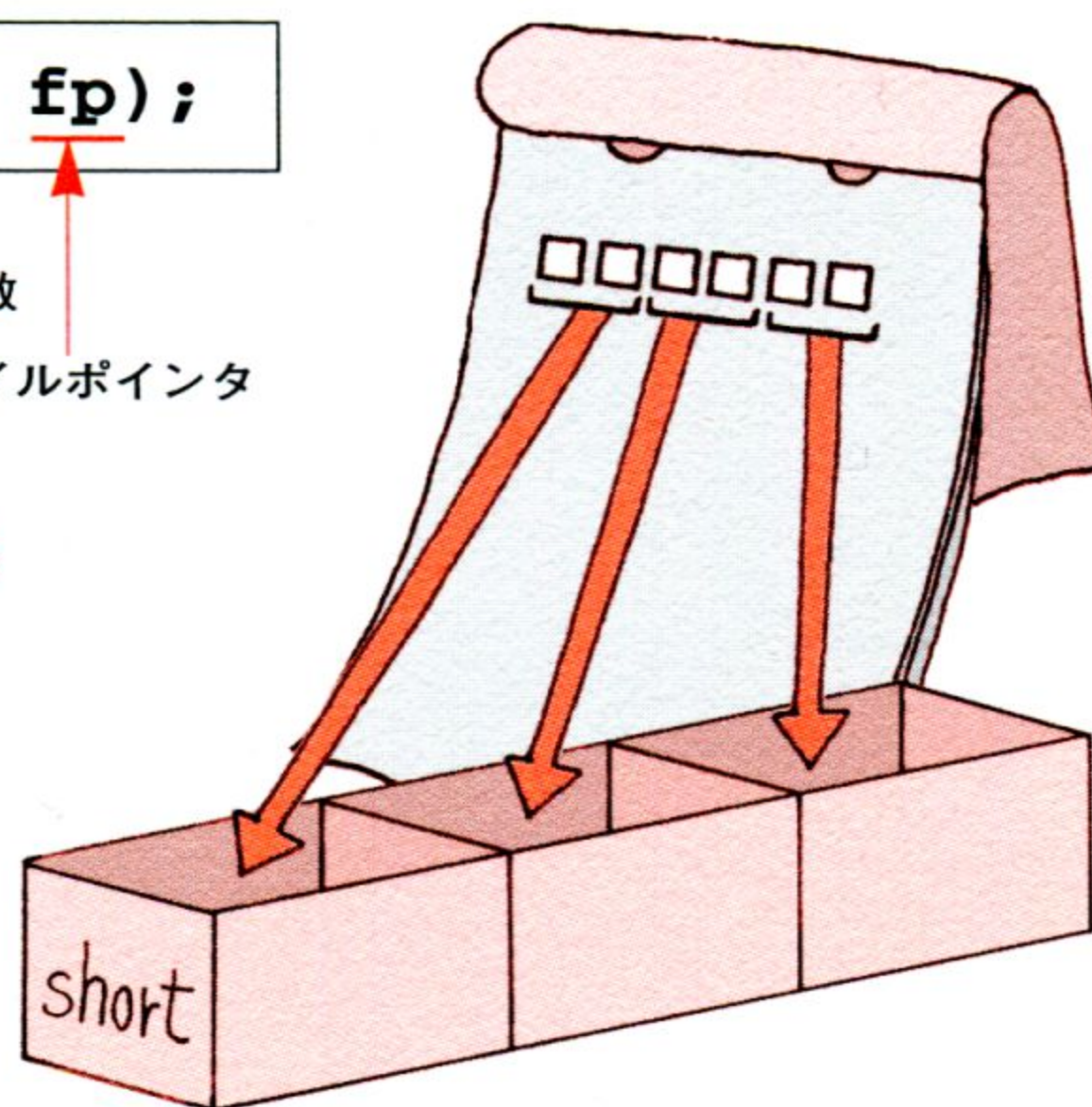
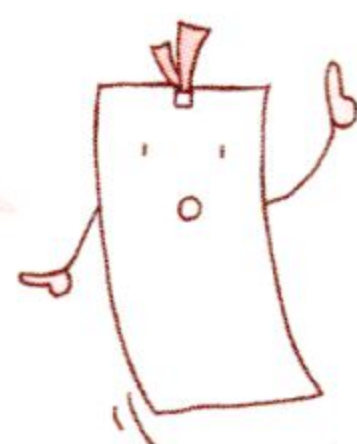
読み込んだデータを格納するバッファ
sizeof(short)=2なので、6バイト分のバッファになります。

②データを読み込む

バイナリデータを読み込むときは、**fread()**（エフリード）関数を使います。
下の例では、fpが示す位置から2バイトのデータを3回分読み込みます。

```
fread(buf, sizeof(short), 3, fp);
```

- buf：バッファの先頭アドレス
- sizeof(short)：読み込みの基本単位（バイト）
- 3：読み込み回数
- fp：ファイルポインタ

short型の変数に3回読み込む場合、基本単位をsizeof(short)、回数を3と指定します。

fread()関数は、実際に読み込めた回数を返します。エラーのときは引数で指定した回数と戻り値が一致しません。

③ファイルを閉じる

最後にfclose()関数を使ってファイルを閉じます。

```
fclose(fp);
```

バイナリの読み書き(2)

読み込みに続いて、書き込みの方法も紹介しましょう。

バイナリファイルの書き出し手順

file4.datにshort型変数3つ分のデータを書き出します。

①ファイルを開く

オープンモードをバイナリ書き出し専用の"wb"にしてファイルを開きます。

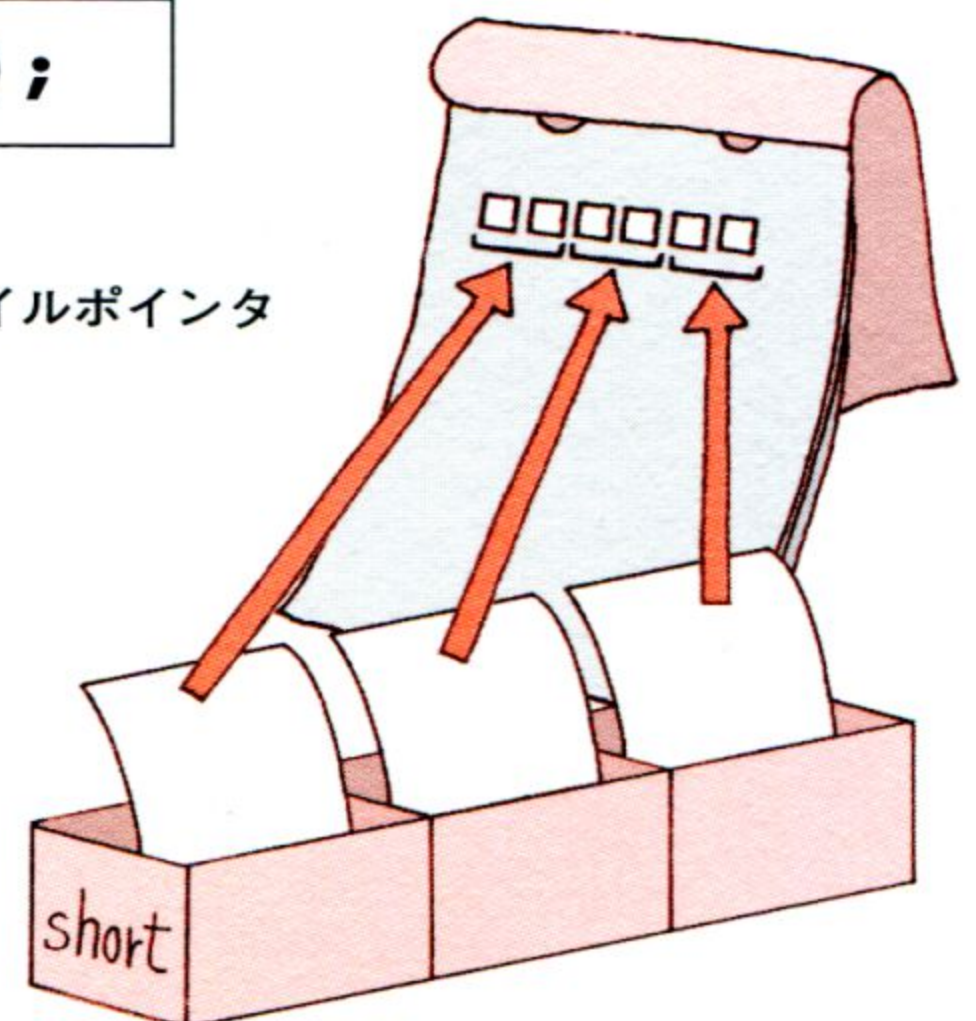
```
short buf[] = {
  0x10,0x20,0x30
}
FILE *fp;
fp = fopen("file4.dat","wb");
```

書き出すデータをあらかじめ用意しておきます。

②ファイルに書き出す

バイナリファイルへのデータの書き出しには**fwrite()**（エフライト）関数を使います。
下の例では、fpの示す位置に2バイト分のデータを3回で書き出します。

```
fwrite(buf, sizeof(short), 3, fp);
```

- buf：バッファの先頭アドレス
- sizeof(short)：書き出しの基本単位（バイト）
- 3：書き出し回数
- fp：ファイルポインタ

fwrite()関数は、実際に書き出せた回数を返します。エラーのときは引数で指定した回数と戻り値が一致しません。

③ファイルを閉じる

最後にfclose()関数を使ってファイルを閉じます。

```
fclose(fp);
```

例

```
#include <stdio.h>

main()
{
    FILE *fp;
    char filename[] = "bintest.dat";
    int buf_w[10], buf_r[10];
    int i;

    for(i = 0; i < 10; i++)
        buf_w[i] = (i+1) * 10;

    if(!(fp = fopen(filename, "wb")))
        return;
    if(fwrite(buf_w, sizeof(int), 10, fp) != 10) {
        fclose(fp);
        return;
    }
    fclose(fp);

    if(!(fp = fopen(filename, "rb")))
        return;
    if(fread(buf_r, sizeof(int), 10, fp) != 10) {
        fclose(fp);
        return;
    }
    fclose(fp);

    for(i = 0; i < 10; i++)
        printf("%d ", buf_r[i]);
}
```

書き出し用のデータを生成

書き出し

読み込み

読み込んだ内容を表示

読み込み、書き出し時のエラー対策も忘れずに！

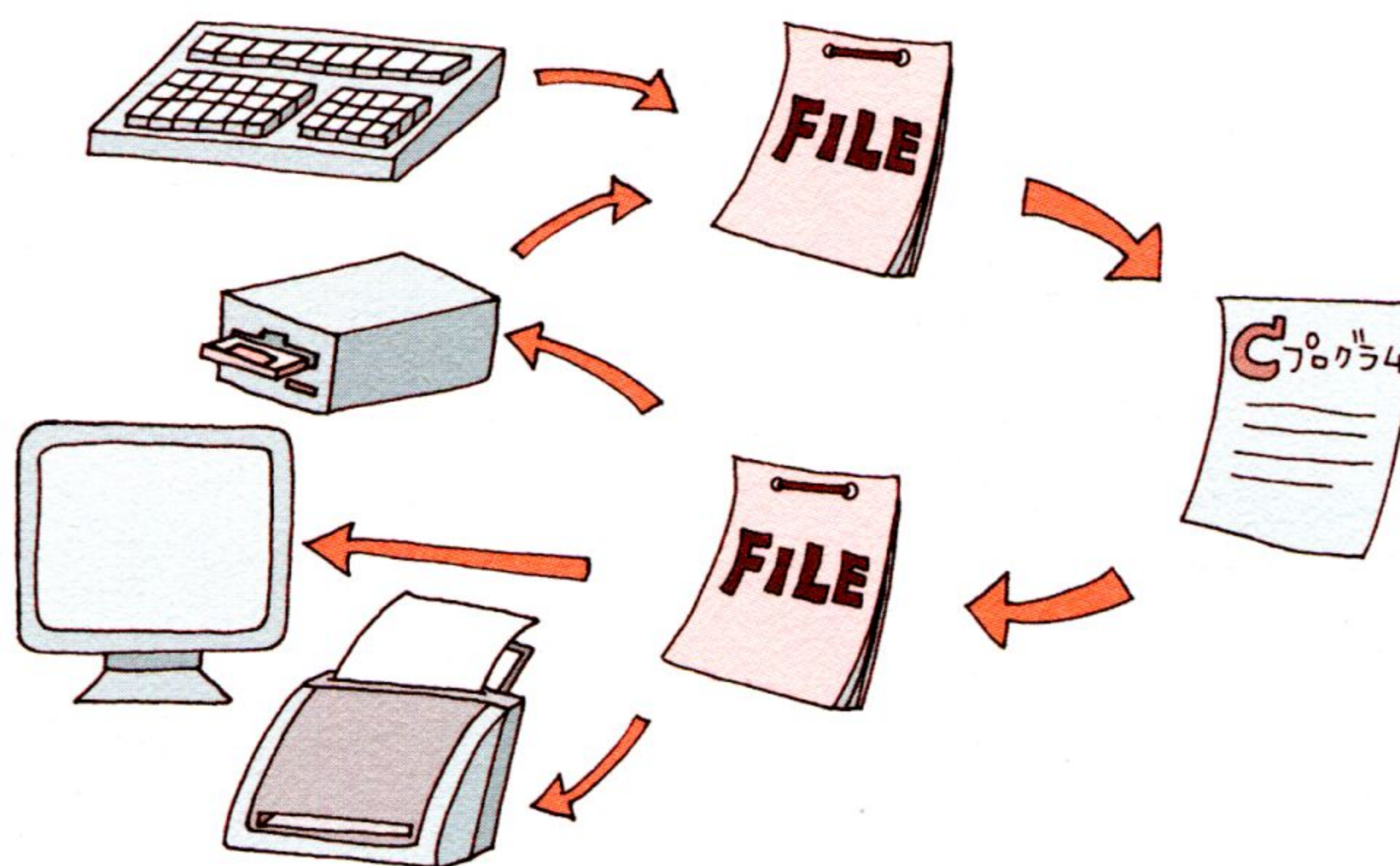
実行結果

```
10 20 30 40 50 60 70 80 90 100
```

一般的な入出力

ファイルは常にディスク上にあるもの…とは限りません。

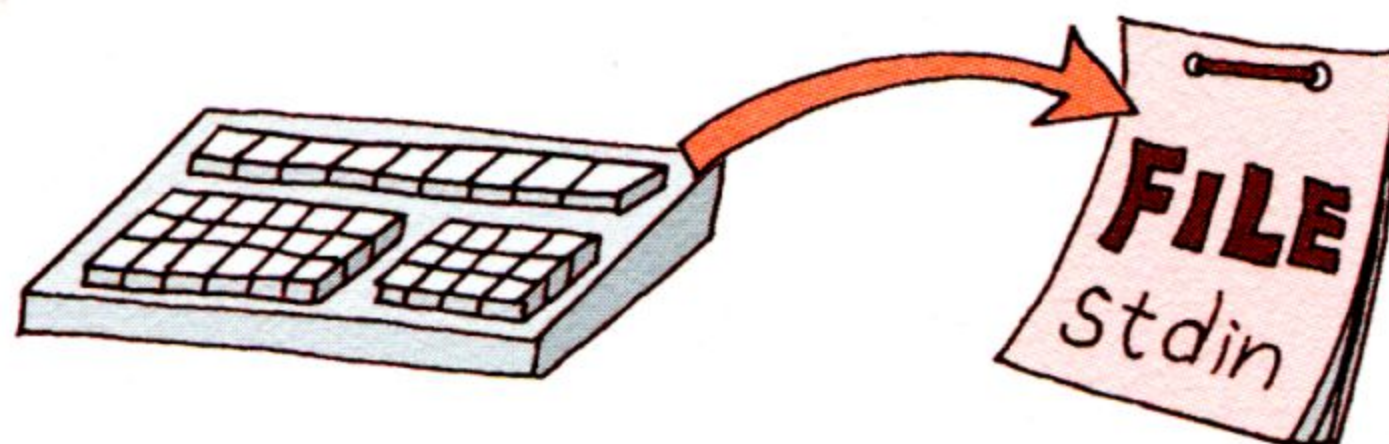
C言語の入出力

すでに見てきたように、C言語では、ファイルポインタを通してディスク上のファイルの入出力を行います。しかし、キーボードやディスプレイなどの入出力装置とのやり取りについても、それらの装置をファイルと見なし、同様にやり取りすることができます。

標準入出力ファイルの種類

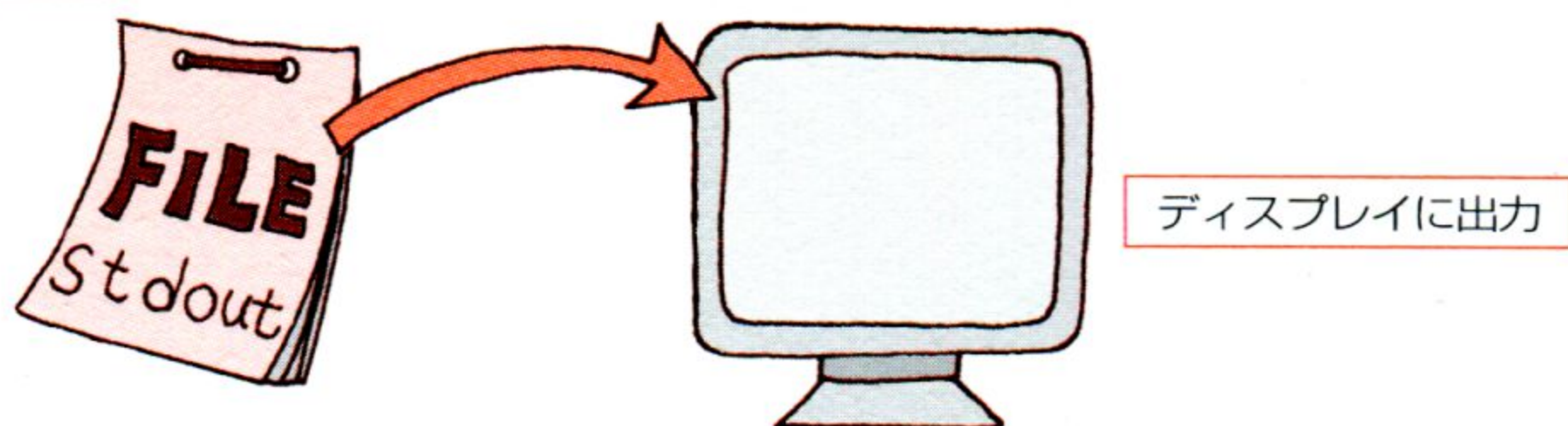
C言語は基本的な入出力のために、**stdin**（スタンダードイン）、**stdout**（スタンダードアウト）、**stderr**（スタンダードエラー）という3つのファイルポインタを用意しています。これらはプログラム実行開始と同時に自動的に開いていて、プログラム上でファイルを開いたり閉じたりする必要はありません。

stdin（標準入力）

stdinは、標準的な入力装置（標準設定ではキーボード）からの入力を受け取るファイルポインタです。

キーボード入力

stdout（標準出力）

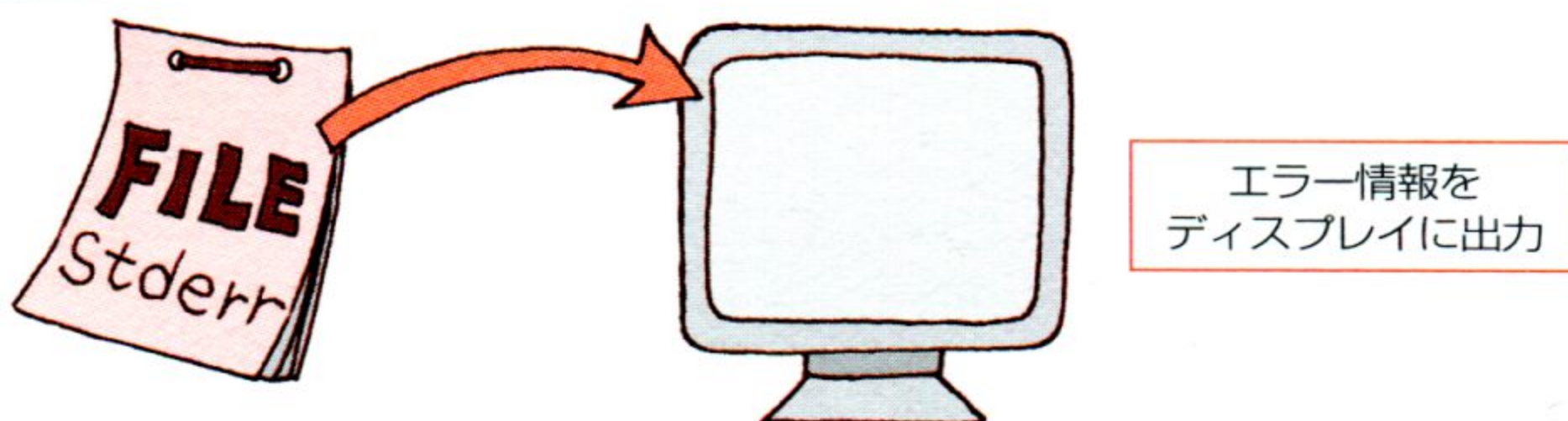
stdoutは、基本的な出力装置（標準設定ではディスプレイ）に出力する際の窓口となるファイルポインタです。

FILE Stdout

ディスプレイに出力

stderr（標準エラー出力）

stderrは、基本的なエラー出力装置（標準設定ではディスプレイ）に出力する際の窓口となるファイルポインタです。

FILE Stderr

エラー情報をディスプレイに出力

ファイル用関数でファイルポインタに標準入出力を指定すると、キーボードやディスプレイからの入出力になります。たとえば、printf()関数はディスプレイに出力する際、データをstdoutに送っているので次のようなことが成り立ちます。

printf("%d", a); ⇔（同じ）⇔ **fprintf(stdout, "%d", a);**

例

```
#include <stdio.h>

main()
{
    char s[30];
    fgets(s, 29, stdin);        ← キーボードから入力した文字をsに格納。
    fputs(s, stdout);           ← sをディスプレイに表示
    fputs("error!¥n", stdout);  ← "error!"をディスプレイに表示。
}
```

実行結果

```
Hello! ↵
Hello!
error!
▌
```

※太字はキーボードから入力した文字

キーボード入力

キーボードで入力したデータを変数や配列に格納する方法を見ていきましょう。

キーボードからのデータ入力

これまでも1文字をキーボードから入力するgetchar()関数を使ってきましたが、ここで、キーボードからのデータ入力に使う主な関数をまとめて紹介します。

scanf()関数（スキャンエフ）

scanf()関数は、キーボードから入力したデータを指定の書式に変換して変数や配列に格納します。

```
int a;
scanf("%d", &a);
```

"%d"：入力データの書式指定
&a：データ格納先のアドレス

&をつけてアドレスにすることに注意。

文字列の場合

```
char s[30];
scanf("%s", s);
```

配列名は配列の先頭要素のポインタとなるため、&は必要ありません。

複数のデータを一度に入力することもできます（入力文字はスペースで区切ります）。

```
int a;
char s[30];
scanf("%d %s", &a, s);
```

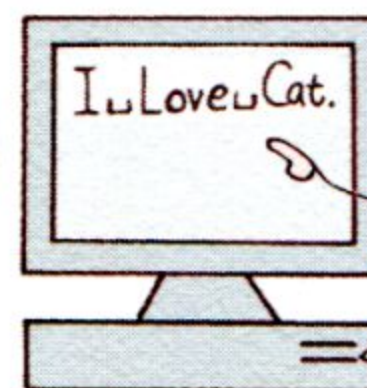
入力文字をスペースで区切るため、スペースを含む文字列を正しく読み込むことはできません。また、入力文字と書式指定があっている保証がないので、あまりお勧めできません。

gets()関数（ゲットエス）

gets()関数は、キーボードから入力した1行分の文字列を文字配列に格納します。
スペースも読み込めます。

```
char s[30];
gets(s);
```

char s[30];：格納用文字配列

I␣Love␣Cat.

スペースも入ります。

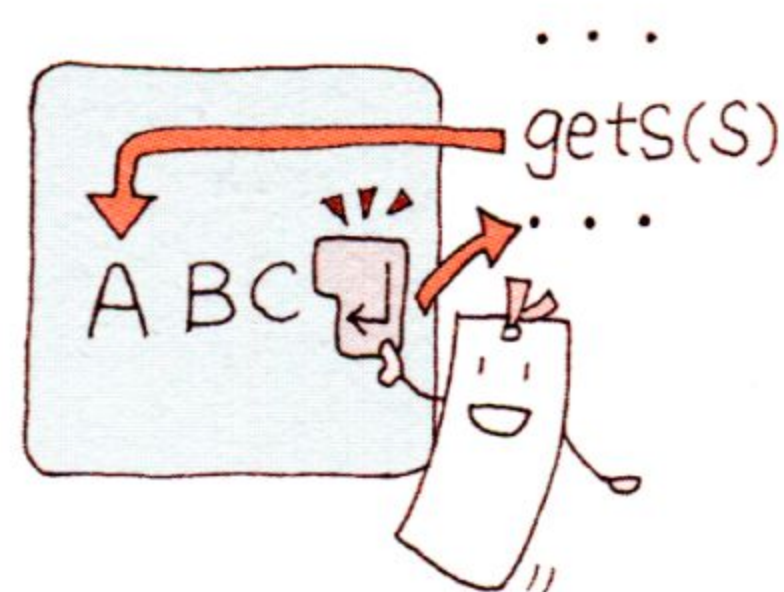
getchar()関数

getchar()関数は、キーボードから入力した文字の1文字だけを変数に格納します。

```
int c;
c = getchar();
```

格納用変数(int型にします。)

gets()関数などを実行すると、プログラムはキーボードからの入力待ち状態になります。
入力後に［enter］キーを押すと、データを受け取ります。

例

```
#include <stdio.h>

main()
{
    int a, b = 7;
    char s[40];
    printf("名前を入力してください¥n");
    gets(s);
    printf("数あてクイズ！0から9の数字を入力してね¥n");
        while(a != b) {
            scanf("%d",&a);
            if((a == b-1) || (a == b+1))
                printf("おしい！¥n");
            else if(a > b+1)
                printf("もっと小さい数です¥n");
            else if(a < b-1)
                printf("もっと大きい数です¥n");
        }
    printf("正解！%sさん、おめでとう！！¥n",s);
}
```

実行結果

```
名前を入力してください
Kobayashi Maiko↵
数あてクイズ！0から9の数字を入力してね
6↵
おしい！
7↵
正解！Kobayashi Maikoさん、おめでとう！！
```

※太字はキーボードから入力した文字

サンプルプログラム

■ファイル中の文字列の置き換え

dog.txtというテキストファイルの"dog"という文字列をすべて"rabbit"に変換し、rabbit.txtという名前で保存します。

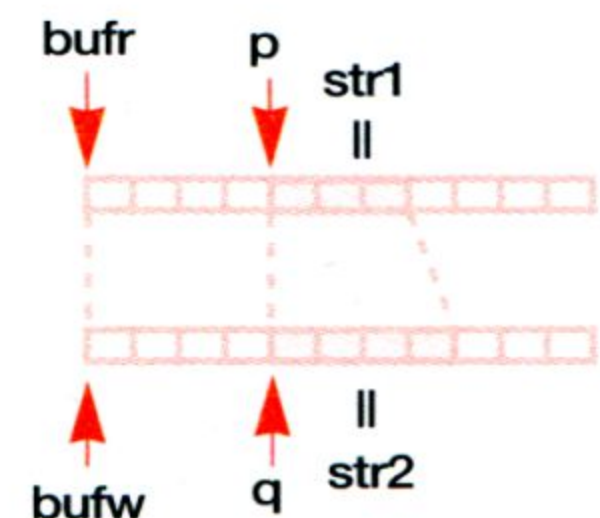
ソースコード

dog.txtの内容

The quick brown fox jumps over the lazy dog.
I like cat and dog.

```
#include <stdio.h>
#include <string.h>

int main()
{
    FILE *fpr, *fpw;              /* 読み/書きファイルポインタ */
    char bufr[256], bufw[256]; /* 読み/書きバッファ */
    char str1[] = "dog";          /* 置換元文字列 */
    char str2[] = "rabbit";       /* 置換後文字列 */
    char *p, *q;

    if(!(fpr = fopen("dog.txt", "r"))) {
        printf("読み込みファイルのオープンに失敗しました。");
        return 1;
    }
    if(!(fpw = fopen("rabbit.txt", "w"))) {
        printf("書き込みファイルのオープンに失敗しました。");
        return 1;
    }
    while(1) {
        fgets(bufr, 256, fpr);
        strcpy(bufw, bufr);
        p = strstr(bufr, str1);
        if(p) {
            q = bufw + (p - bufr);
            strcpy(q, str2);
            strcpy(q+strlen(str2), p+strlen(str1));
        }
        fprintf(fpw, "%s", bufw);
        if(feof(fpr))
            break;
    }
    fclose(fpr);
    fclose(fpw);
    return 0;
}
```

strstr()関数
第1引数の中から第2引数の文字列を探し、その位置のポインタを返す。

文字列を置き換える。

実行結果

(rabbit.txtの中身)

```
The quick brown fox jumps over the lazy rabbit.
I like cat and rabbit.
```

■dumpコマンドの作成

バイナリファイルの中身を表示するプログラム"dump"を作ってみましょう。

ソースコード

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    FILE *fp;
    unsigned char buf[16];  /* 読み込みバッファ */
    unsigned long addr = 0; /* 先頭からのアドレス */
    int readnum, i;

    if(argc <= 1) {
        printf("usage:dump filename¥n");
        return 1;
    }
    if(!(fp = fopen(argv[1], "rb"))) {
        printf("file open error.¥n");
        return 1;
    }
    while(1) {
        printf("%08X  ", addr);
        readnum = fread(buf, 1, 16, fp);
        /* バイナリデータの表示 */
        for(i = 0; i < readnum; i++) {
            if(i == 8)
                printf(" ");
            printf("%02X ", buf[i]);
        }
        for(i = readnum; i < 16; i++) {
            if(i == 8)
                printf(" ");
            printf("   ");
        }
        printf(" ");
        for(i = 0; i < readnum; i++)
            printf("%c", (32 <= buf[i] && buf[i] <= 126) ? buf[i] : '.');
        printf("¥n");
        addr += 16;
        if(feof(fp))
            break;
    }
    fclose(fp);
    return 0;
}
```

readnumは実際に読み込んだバイト数。

制御コードは"."に置き換える。

実行結果

```
>dump bintest.dat
00000000  0A 00 00 00 14 00 00 00  1E 00 00 00 28 00 00 00  ............(...
00000010  32 00 00 00 3C 00 00 00  46 00 00 00 50 00 00 00  2...<...F...P...
00000020  5A 00 00 00 64 00 00 00                           Z...d...
```

※太字はキーボードから入力した文字です。また、この実行結果は、117ページの例題で作ったbintest.datを用いたときのものです。

COLUMN
コラム

〜fseek()関数〜

この章では主にファイルのデータを読み書きする関数を学んできました。ファイルの読み書きはファイルポインタの位置からはじまり、その位置はファイルのオープンモードによって決まるということも紹介しました。そこで、「ファイルの読み書きする位置は自分では決められないの？」という疑問を持った人もいるのではないでしょうか。実はファイルポインタの位置を移動するための関数も用意されています。

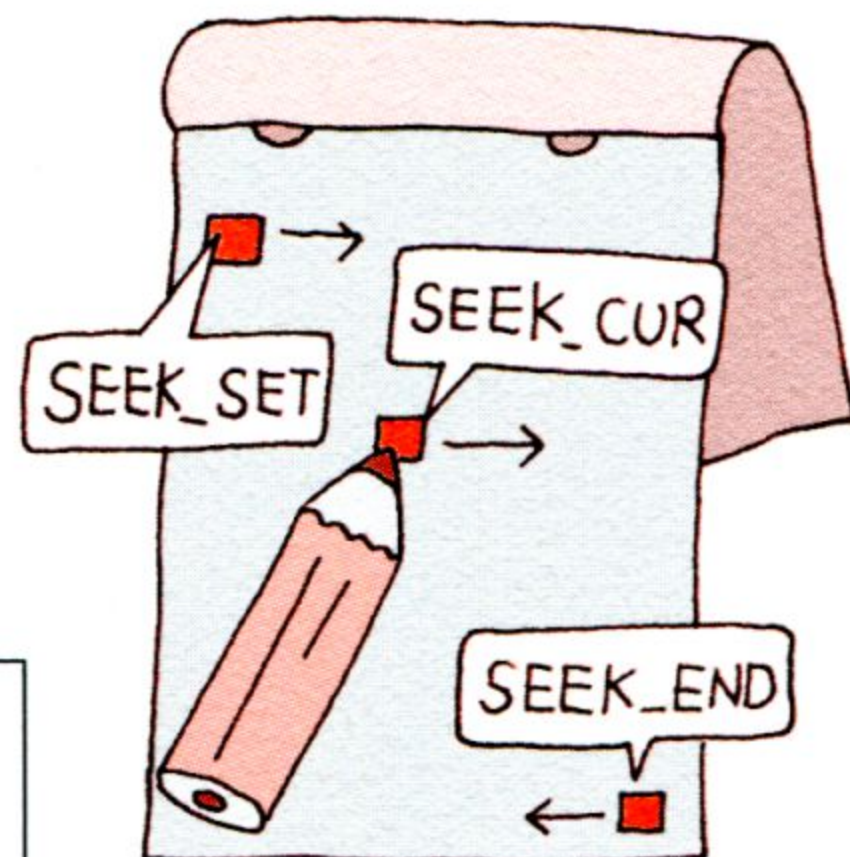
fseek()関数は、ファイルポインタを移動するときに使います。

```
fseek(fp, 10, SEEK_SET);
```

ファイルポインタ

移動するバイト数

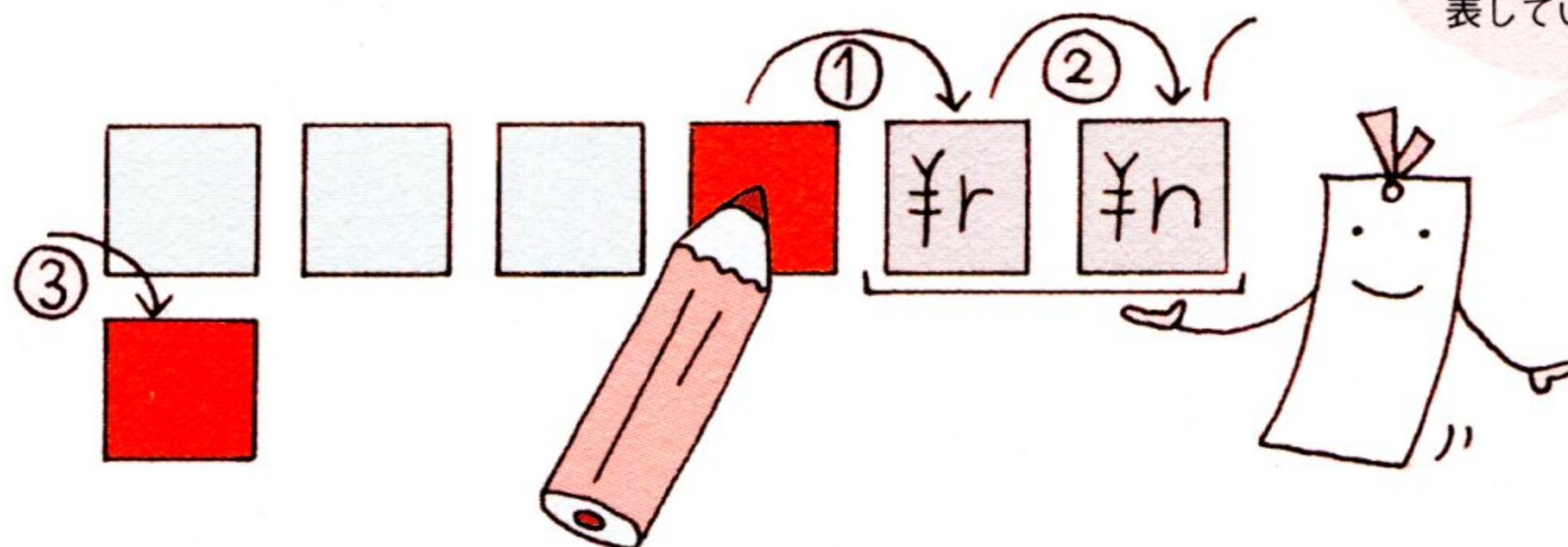
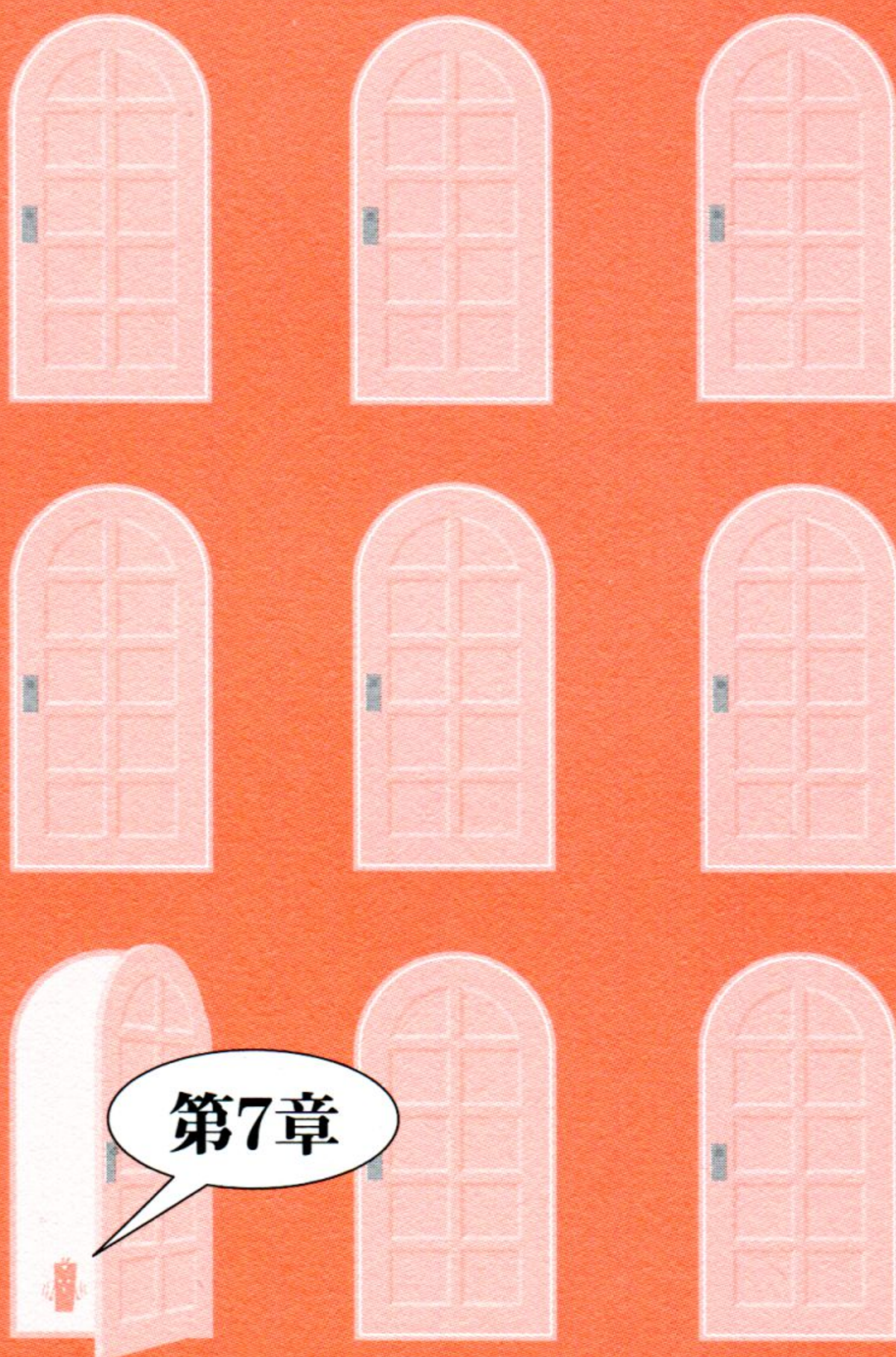
シークモード
ファイルポインタの移動開始地点を指定します。

SEEK_SET・・・ファイルの先頭
SEEK_CUR・・・ファイルポインタの現在位置
SEEK_END・・・ファイルの最後

fseek()関数では、テキスト・バイナリどちらのファイルでも、1バイトずつ進みます。テキストファイルでは、開発環境により改行コードが2バイトになることがありますが、それぞれ1バイトと扱うので注意しましょう。

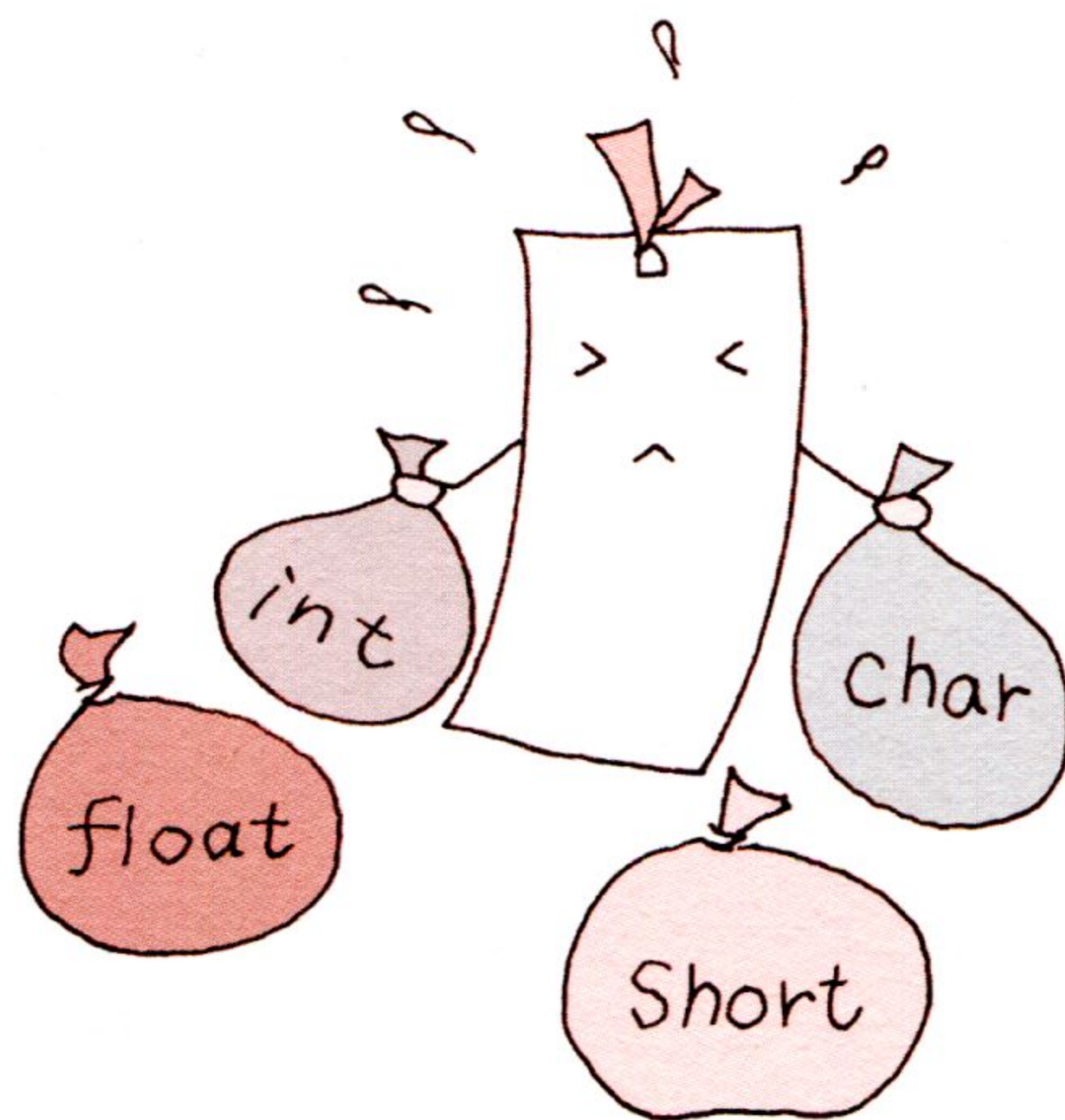
7 構造体

第7章

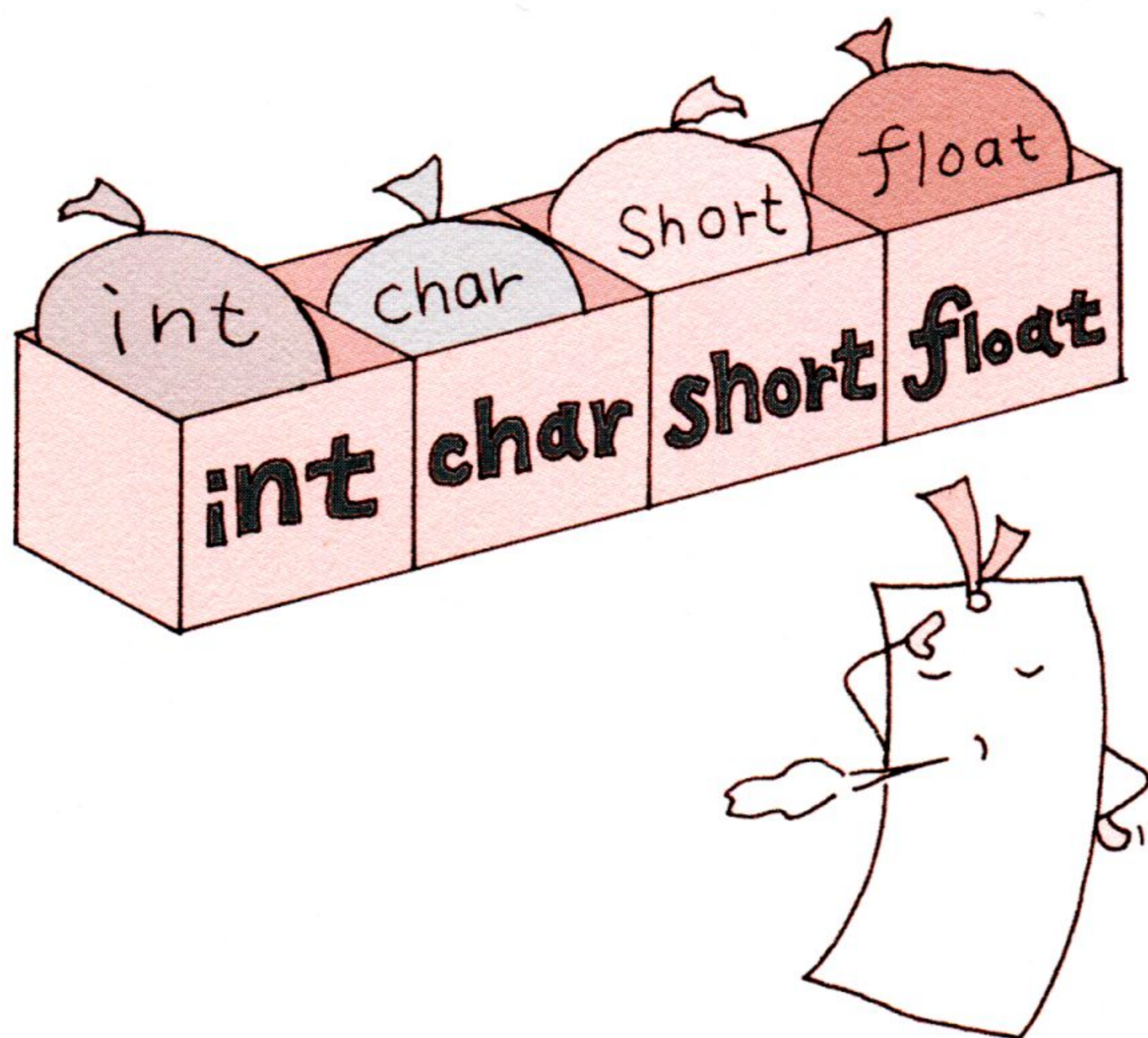
第7章は ここがkey!

Topics

データを管理しやすくまとめてみよう！

第7章では**構造体**を紹介します。随分と仰々しい名前ですが、さて構造体とは一体何なのでしょう？

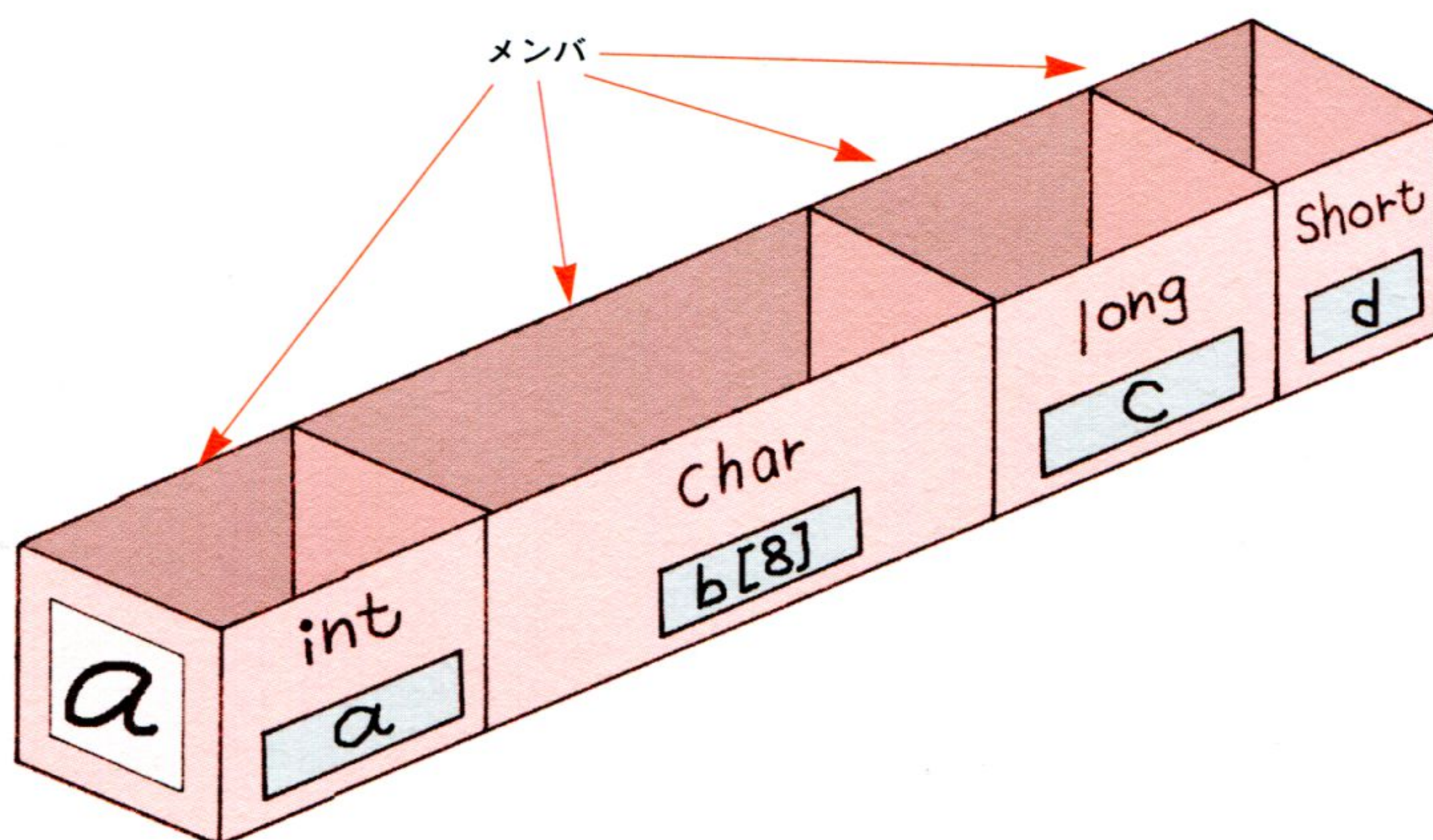
構造体について説明する前に、第4章で学んだ配列を思い出してください。配列とは「同じ型」のデータをひとまとめにしたものでした。これに対し、構造体は「違う型」のデータをひとまとめにしたものです。みなさんがこれから、より複雑なプログラムを作れるようになると、「違う型だけど関係のあるデータだからまとめておきたいな……」と思うことがあると思います。そんなときこそ構造体を使います。

まずは構造体で、どのような型の変数をまとめるのかを指定します。これを構造体の**テンプレート**といいます。テンプレートを定義しただけではその中にデータを入れることはできません。そこで次に、その構造体の型を持つ変数（**構造体変数**）を用意します。

ちょうど、溶かしたチョコレートをあらかじめ用意した型に流し込んで固めるようなイメージです。そうして出来上がった構造体変数の箱にデータをまとめて入れておくことで、膨大なデータをすっきりとまとめることができるのです。

さらに応用として、構造体を「配列」にした**構造体配列**についても紹介します。構造体配列は、構造体の型に流し込んでできた箱と同じものを指定の数だけ用意したものです。これは社員名簿などに使えそうですね。たとえば記録するデータの種類には、社員番号・名前・性別・生年月日・入社年月日という5つの項目があり、社員が1000人いたとします。構造体配列を使えば、まず5つの項目が入るテンプレートを用意し、構造体配列で1000個の同じ形の箱を作ればよいわけです。

構造体

構造体とは何か、宣言するにはどうすればよいかを見ていきます。

構造体の概念

構造体とは複数の型の変数をひとまとめにしたものです。配列に近いイメージですが、構造体は異なる型でも、配列であっても1つにまとめることができます。
また、構造体によってまとめた要素ひとつひとつのことを**メンバ**といいます。

構造体の宣言

構造体の宣言は次の2ステップで行います。

①構造体テンプレートの宣言

どのような型の変数や配列をまとめるかという枠組みを決めます。

②構造体変数の宣言

データを実際に記憶するために、構造体のテンプレートを持った変数を用意します。

なお、構造体テンプレートと構造体変数の宣言を一度に記述することもできます。

»構造体テンプレートの宣言

どのような変数を構造体として1つにまとめるのか定義することを「構造体テンプレートの宣言」といいます。宣言は次のように行います。

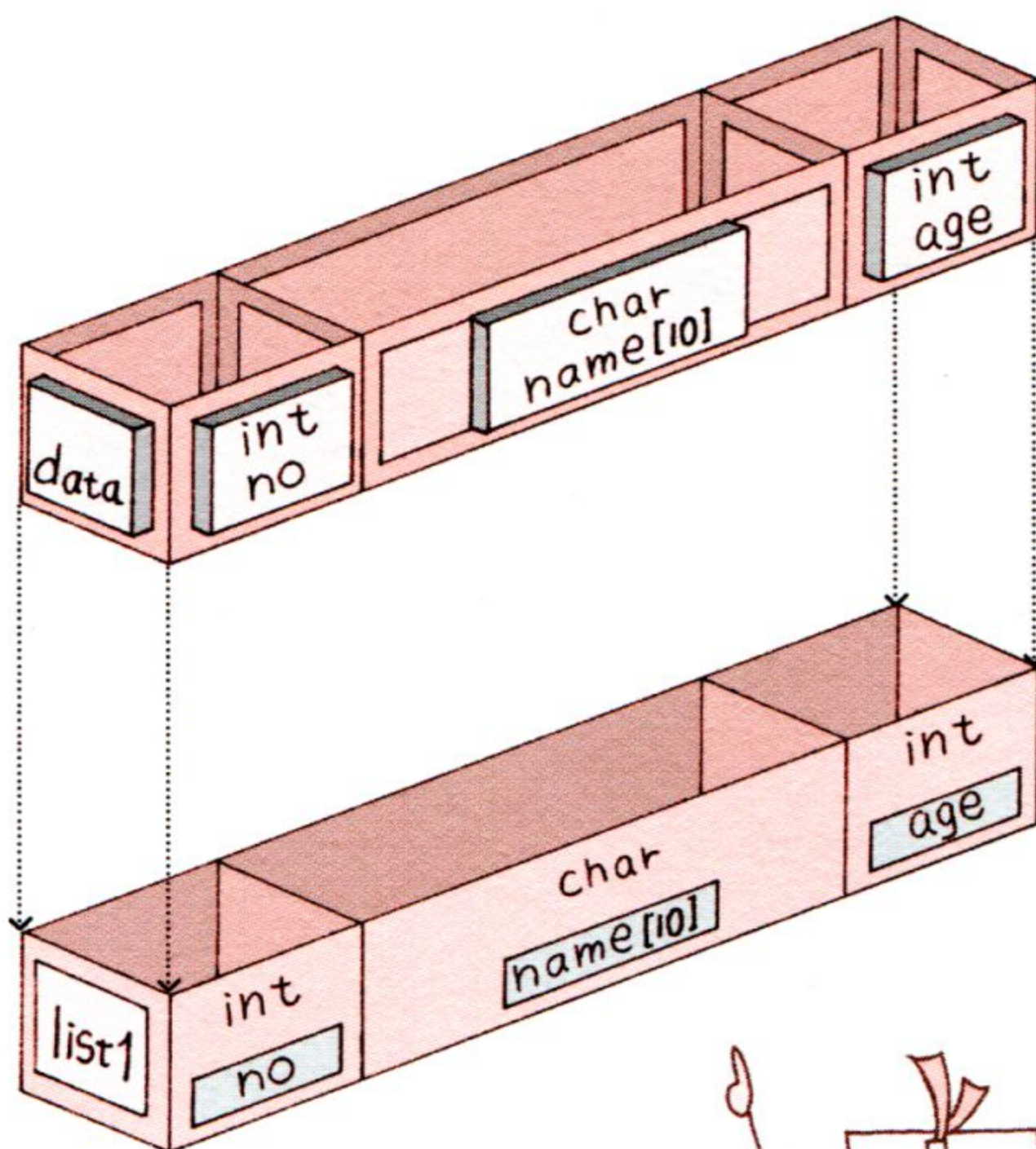
構造体テンプレート名

```
struct data{
    int no;
    char name[10];
    int age;
};
```

メンバ
構造体を構成する要素です。
;で区切って列挙します。

»構造体変数の宣言

実際に構造体を使うには構造体の型を持った変数（構造体変数）を用意します。
宣言は次のように行います。

```
struct data list1;
```

構造体テンプレート名　構造体変数名

dataのテンプレートを持ったlist1という構造体変数を用意しています。

»構造体テンプレートと構造体変数を同時に宣言

次のように枠組みと変数の宣言を同時に行うこともできます。

```
struct data{
    int no;
    char name[10];
    int age;
} list1;
struct data list2;
```

構造体テンプレート名
同時宣言の場合、省略可能ですが、指定するとあとで再利用できます。

メンバ

構造体変数名

あとから構造体変数を追加できます。

構造体の活用

構造体のそれぞれのメンバに値を入れる方法、値を参照する方法を見ていきます。

構造体の初期化

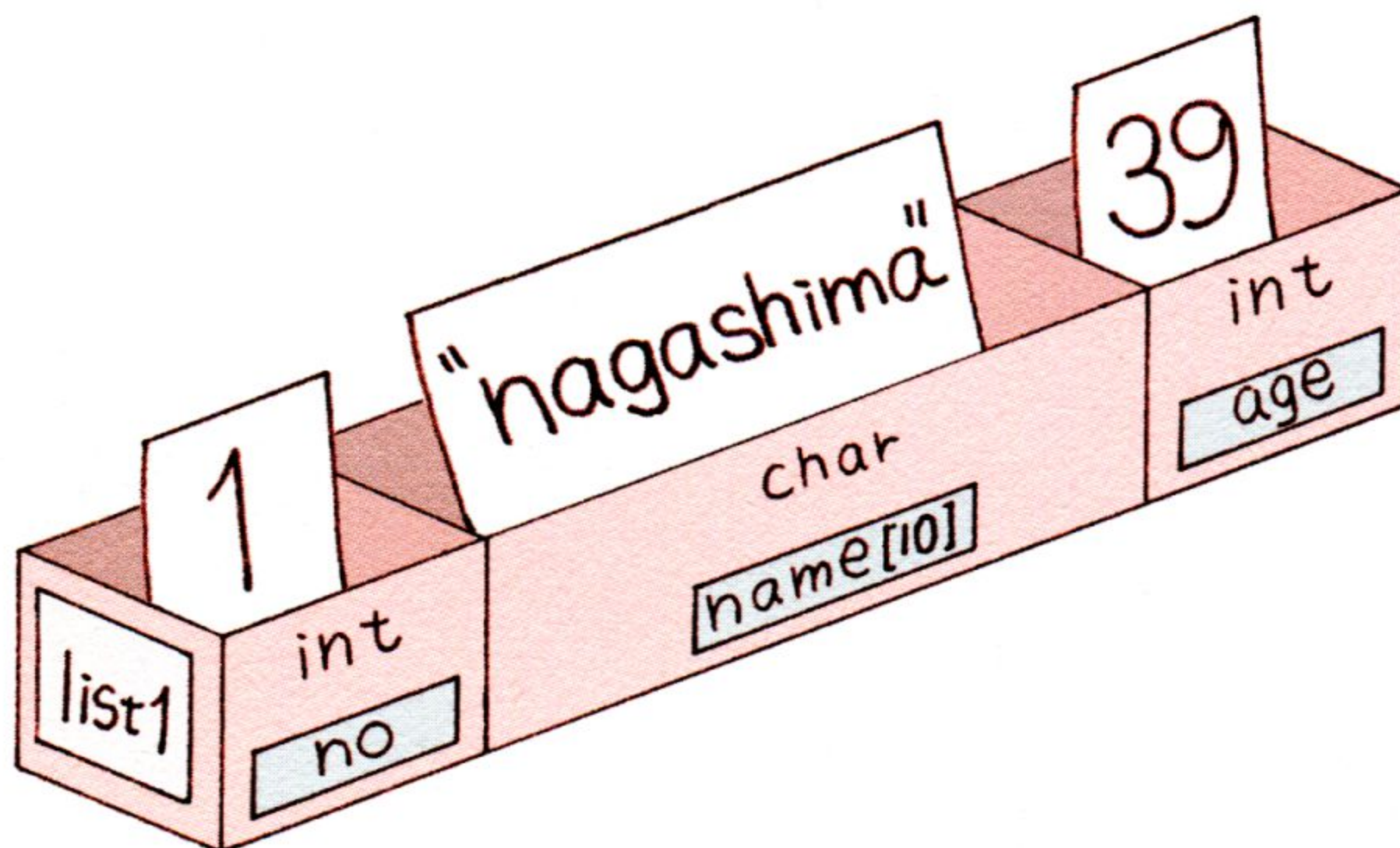
構造体変数の初期化は宣言時に次のように行います。

```
struct data{
     int no;
     char name[10];
     int age;
};
struct data list1 = {1, "nagashima", 39};
```

構造体dataを宣言

初期化

構造体テンプレート名 **構造体変数名**

初期化リスト
宣言に合わせてデータを記述します。

メンバが多いときは、折り返して書くこともあります。

```
struct data list1 = {
     1,
     "nagashima",
     39
};
```

宣言と混同しないように！

構造体メンバへのアクセス

構造体変数のメンバを参照するには、「.（ピリオド）」を使って、どのメンバを参照するのか指定します。

ピリオド

```
printf("%d %s %d¥n", list1.no, list1.name, list1.age);
```

構造体変数名　メンバ名

変数名とメンバ名をピリオドでつなぎます。

構造体変数へデータを代入するときも同様です。

```
list1.no = 3;
strcpy(list1.name , "nagashima”);
list1.age = 39;
```

構造体変数list1の、
メンバnoに3を代入
メンバnameにnagashimaをコピー
メンバageに39を代入

例

```
#include <stdio.h>

struct _point2d {
    double x;
    double y;
} pt;

main()
{
    pt.x = 30.0;
    pt.y = 23.6;
    printf("pt = (%4.1f, %4.1f)¥n", pt.x, pt.y);
}
```

実行結果

```
pt = (30.0, 23.6)
```

基本的なプログラム　演算子　制御文　配列とポインタ　関数　ファイルの入出力　構造体　プログラムの構成

構造体とポインタ

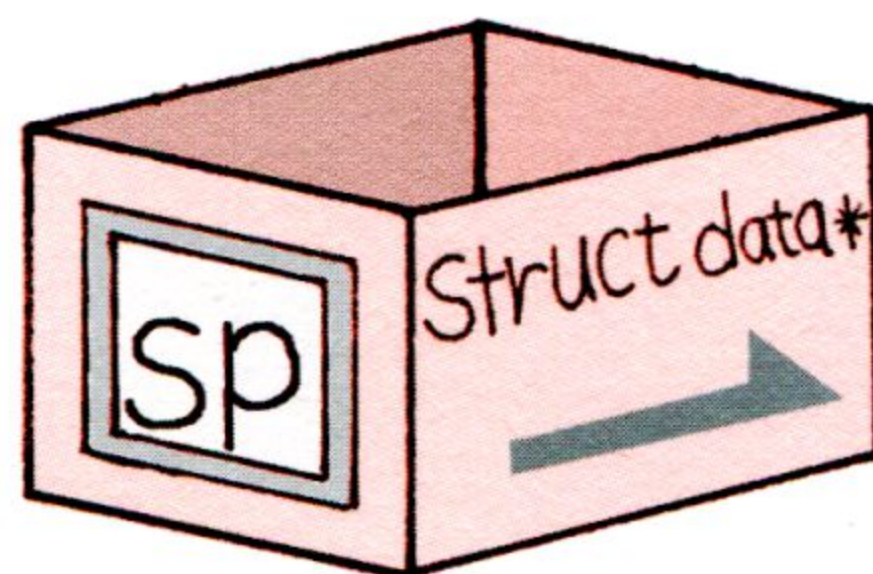
変数に対するポインタと同じように構造体変数に対するポインタを考えることができます。

構造体を指し示すポインタ

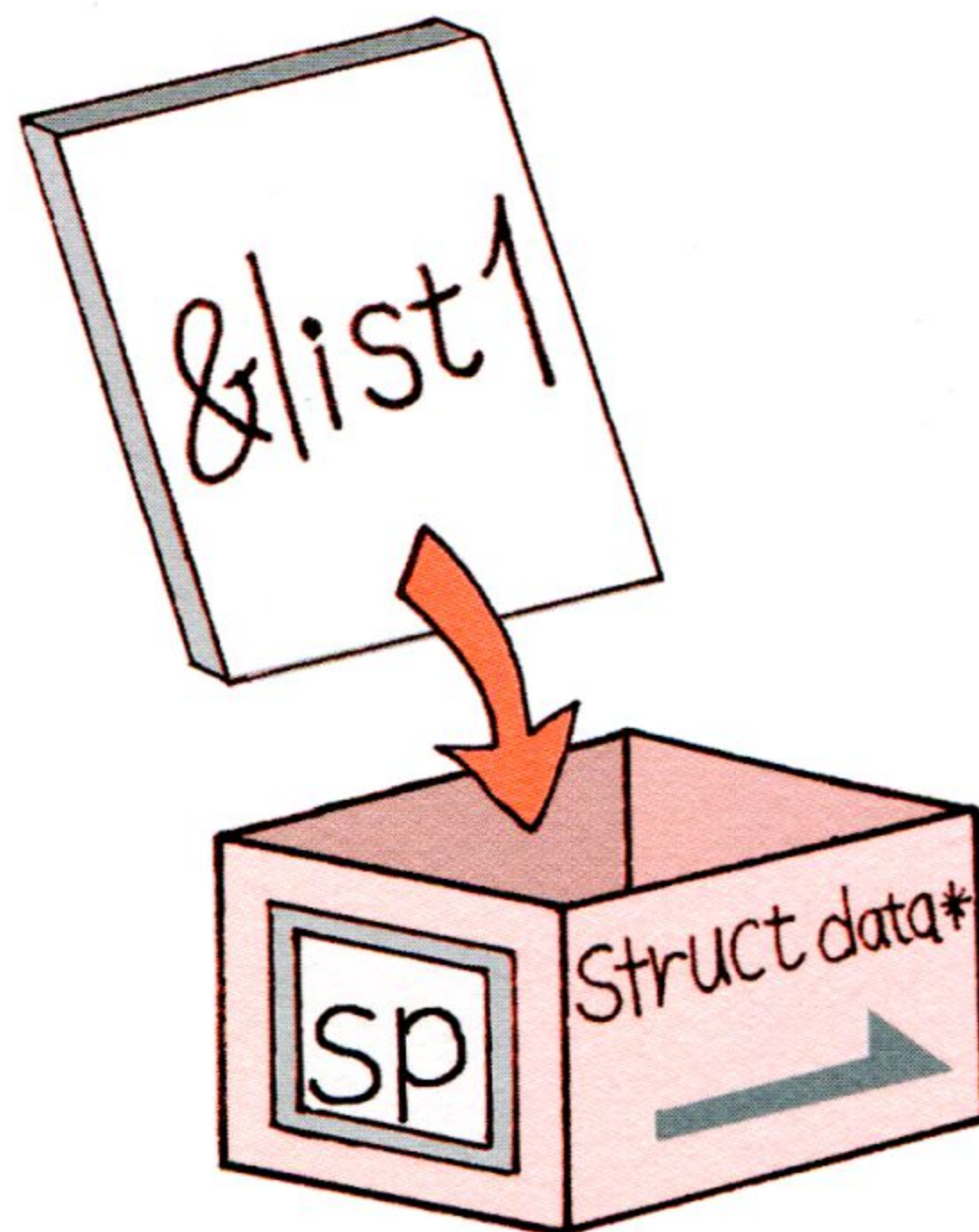
構造体変数をポインタで指し示すことを考えてみましょう。
基本的な考え方は変数に対するポインタと同じです。構造体を指し示すポインタを宣言するには、ポインタ名の前に*をつけます。

実体を宣言するとき使うので、構造体テンプレート名を指定します。

```
struct data {
    int no;
    char name[10];
    int age;
};
struct data *sp;
```

構造体テンプレート名　ポインタ名

ポインタへのアドレスの代入は次のように行います。

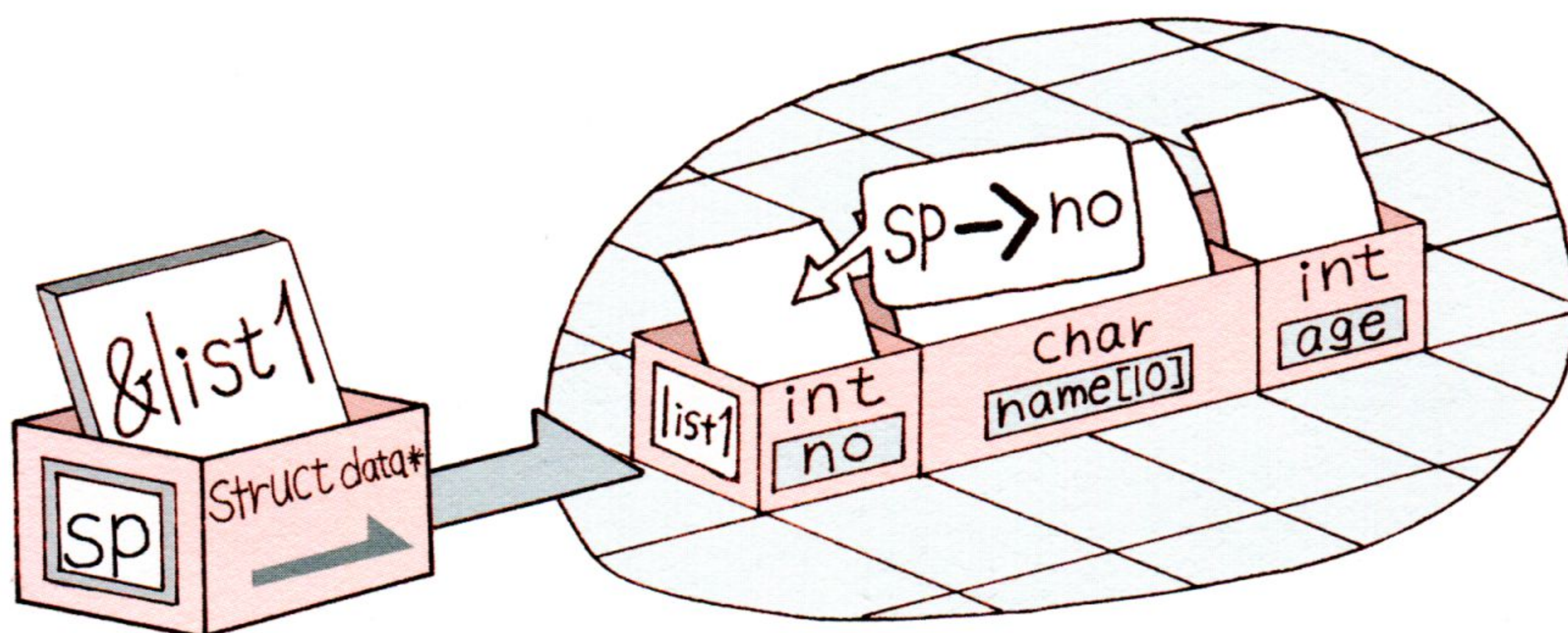
```
struct data list1;

sp = &list1;
```

構造体変数名

ポインタを使った構造体の参照

ポインタを使って構造体のメンバを参照するには -> という記号（**アロー演算子**）を使います。次のように記述します。

アロー演算子

```
printf("%d %c %d¥n", sp->no, sp->name, sp->age);
```

ポインタ名　メンバ名

例

```
#include <stdio.h>

struct _colorpoint2d {
    double x, y;
    int colorid;
} cpt;
struct _colorpoint2d *ppt = &cpt;

main()
{
    ppt->x = 2.4;
    ppt->y = 3.2;
    ppt->colorid = 1;
    printf("(%3.1f, %3.1f) color=%d¥n",
        ppt->x, ppt->y, ppt->colorid);
}
```

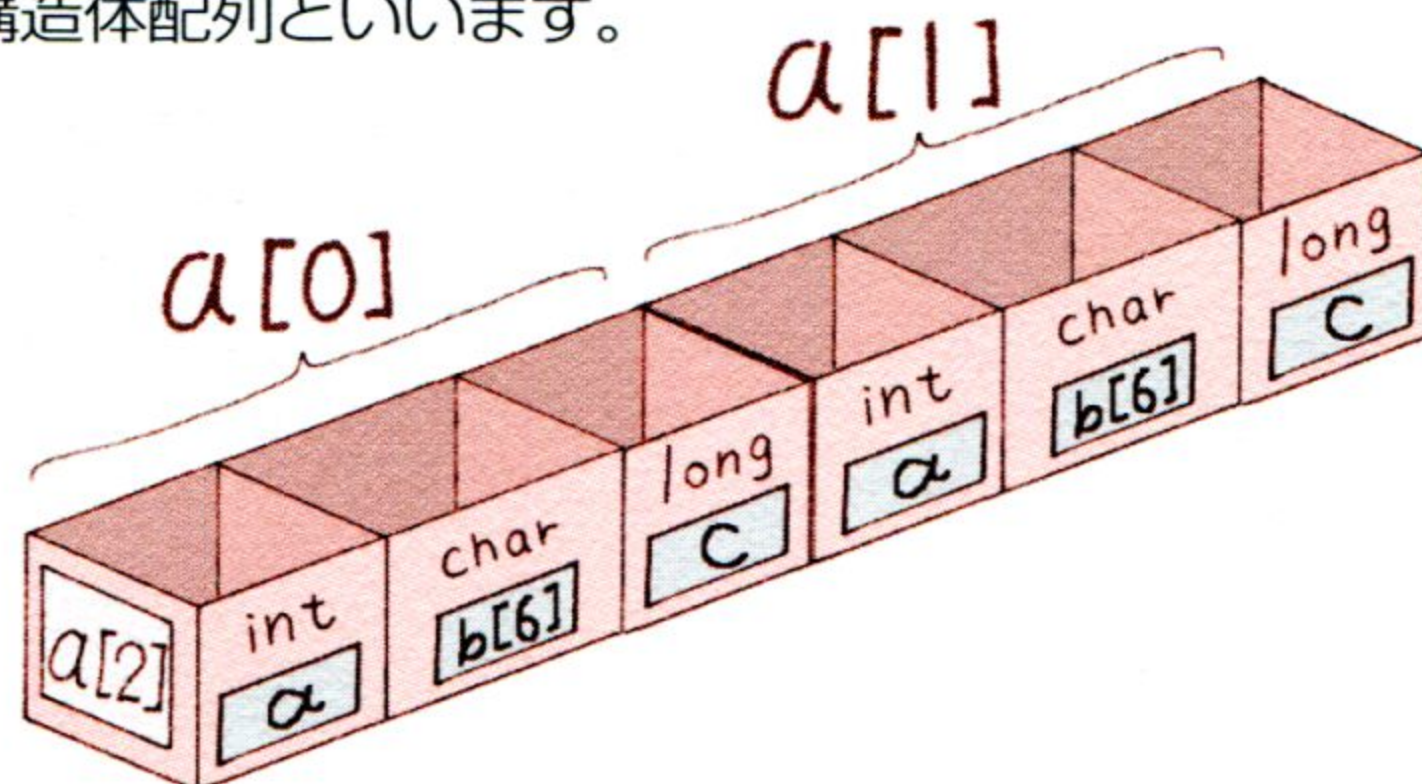
実行結果

```
(2.4, 3.2) color=1
```

構造体と配列

構造体の応用として、構造体配列について見ていきます。名簿のデータベースなどに使えそうです。

構造体配列

通常の変数と同じように、構造体変数についても配列を考えることができます。
これを構造体配列といいます。

構造体配列の使い方は構造体変数とほとんど変わりません。

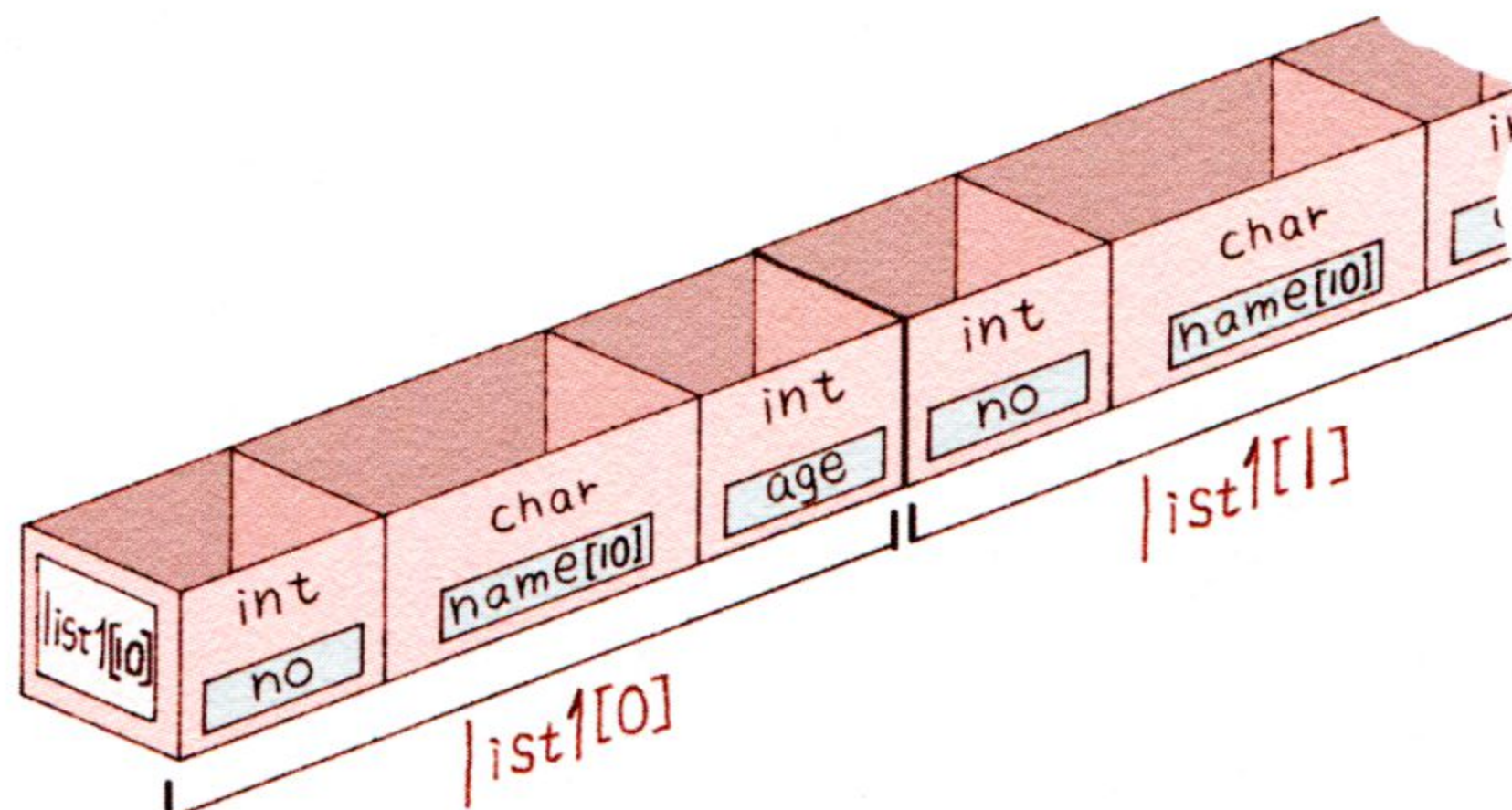
構造体配列の宣言

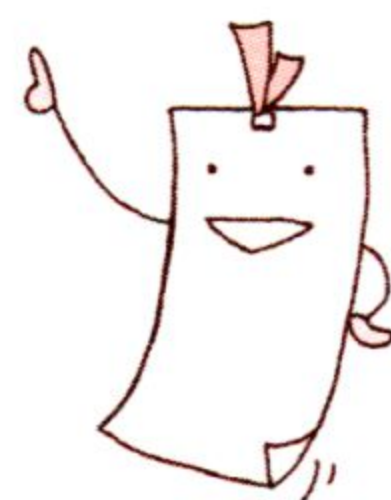
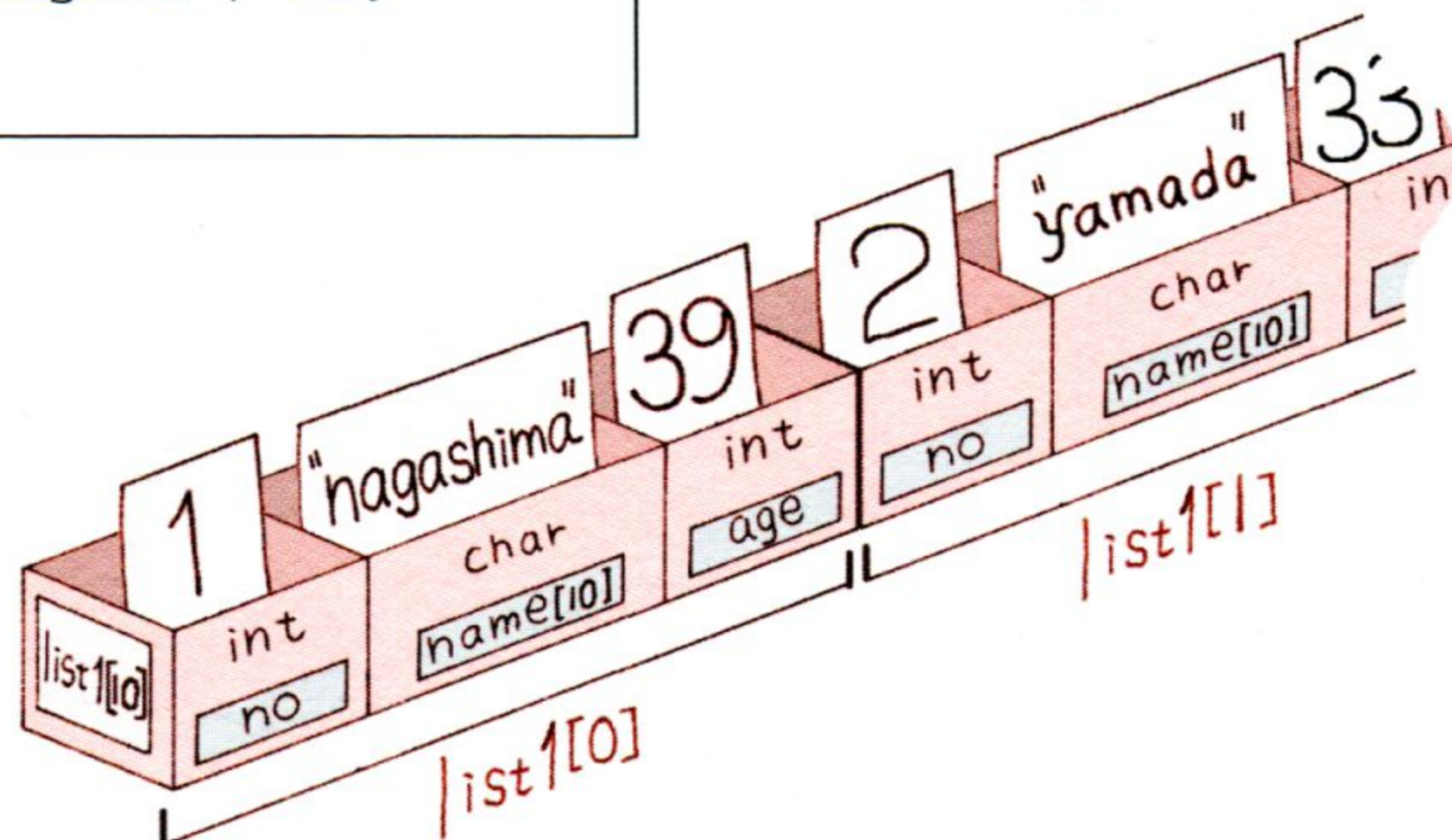
構造体テンプレートと構造体配列を個別に宣言

```
struct data{
    int no;
    char name[10];
    int age;
};
struct data list1[10];
```

構造体テンプレートと構造体配列を同時に宣言

```
struct data{
    int no;
    char name[10];
    int age;
} list1[10];
```

構造体配列

構造体配列の初期化

```
struct data list1[10] = {
    {1, "nagashima", 39},
    {2, "yamada", 33},
            ⋮
    {9, "tonegawa", 31}
};
```

要素ごとに{}でくくります。

構造体配列の参照

次の3つはfor文を使って配列の最後の要素まで参照を繰り返します。どれも、同じ結果を表示します。

[ ]を使った表記

```
int i;
for(i = 0; i < 10; i++)
    printf("%d %s %d¥n", list1[i].no, list1[i].name, list1[i].age);
```

***を使った表記**

```
int i;
struct data *sp = list1;
for(i = 0; i < 10; i++)
    printf("%d %s %d¥n",
        (*(sp+i)).no, (*(sp+i)).name, (*(sp+i)).age);
```

構造体配列名は要素0番となるため、&は必要ありません。

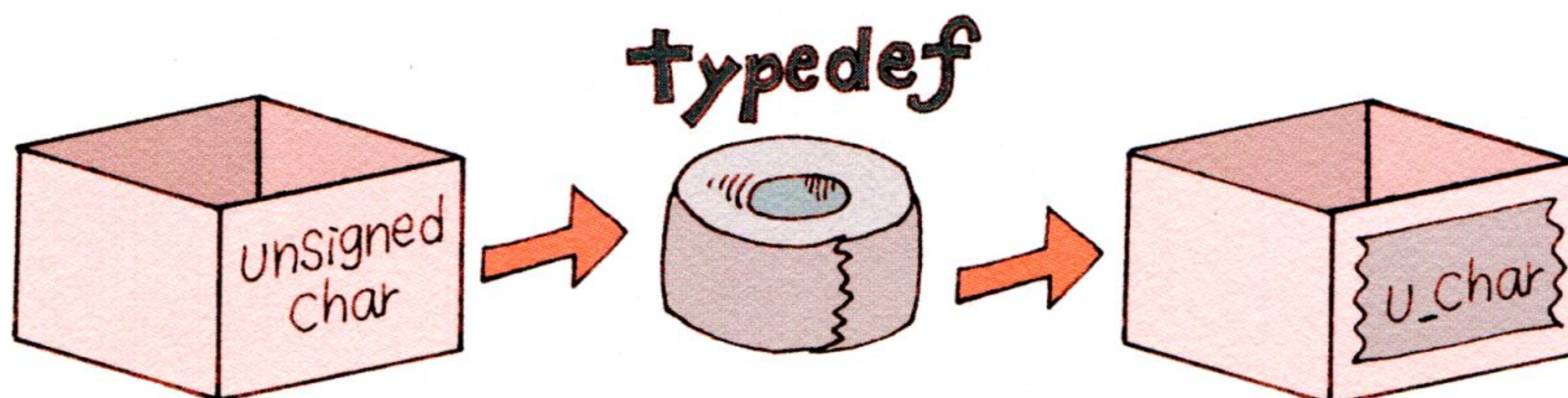
-> を使った表記

```
struct data *sp;
for(sp = list1; sp! = list1+10; sp++)
    printf("%d %s %d¥n", sp->no, sp->name, sp->age);
```

型の再定義

長い型名はプログラムを記述する上で使いづらいものです。typedef(タイプデフ)を使うと簡潔に書けます。

型の名前を変更する

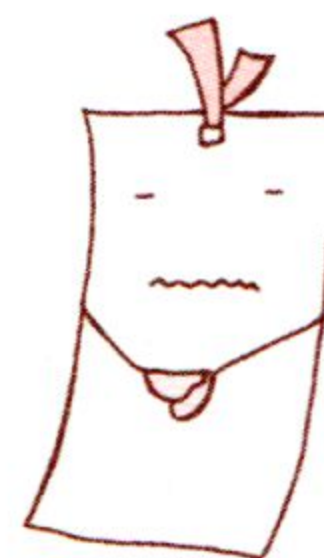
型の名前は、unsignedなどと組み合わせると、長くなります。そんなときは、**typedef** を使って任意の名前をつけることができます（**型の再定義**といいます）。

次のようにすると、「unsigned char」を「u_char」と記述できるようになります。

既存の型名 / 新しい型名

```
typedef unsigned char u_char;

u_char c;    ← unsigned char c と同じ意味です。
```

ポインタ型の再定義は次のようになります。

```
typedef unsigned int *pt_int;

pt_int a;    ← unsigned int *a と同じ意味です。
```

aの前に * は必要ありません。

C 構造体名を変更

typedefを使えば構造体のテンプレートにも任意の名前を定義できます。

構造体名
DATAという名前をつけているので、省略しても再利用に支障ありません。

```
typedef struct data {
    int no;
    char name;
    int age;
} DATA;   ← 新しい名前

DATA list1;
```

DATAは型名なので、structは必要ありません。

どちらも同じ結果になります。

構造体名

```
struct data{
    int no;
    char name;
    int age;
};

struct data list1;
```

構造体名の前にstructが必要です。

例

```
#include <stdio.h>

typedef struct _PROFILE {
    char name[40];
    int age;
} PROFILE;
main()
{
    PROFILE prof[2] = {
            {"Maiko", 20 },
            {"Naoki", 31 }
        };
    int i;
    for(i = 0; i < 2; i++)
        printf("%sさんは %d歳¥n",
            prof[i].name, prof[i].age);
}
```

実行結果

```
Maikoさんは20歳
Naokiさんは31歳
```

サンプルプログラム

■ カロリー計算プログラム

食品の登録ができるカロリー計算のプログラムです。

ソースコード

```
#include <stdio.h>
#include <string.h>

typedef struct _CALORIE {
    char name[40];
    float value;
} CALORIE;

int calregist(CALORIE *, int);
float calcalc(CALORIE *, int);

int main()
{
    CALORIE cal[500] = {
        {"米飯",    150.0},          {"中華麺",  57.1},
        {"そば",    133.3},          {"うどん", 100.0},
        {"素麺",    133.3},          {"食パン", 250.0}
    };
    int cal_num = 6;
    int mode = 0;

    printf("カロリー計算ツール¥n");
    while(1) {
        printf("登録は1を、計算は2を、終了は0を入力してください : ");
        scanf("%d", &mode);
        if(mode == 0)
            break;
        else if(mode == 1)
            cal_num = calregist(cal, cal_num);
        else if(mode == 2)
            printf("総カロリー:%6.2fkcal¥n¥n", calcalc(cal, cal_num));
    }
    return 0;
}

/*******************************************
  calregist() カロリーリストへ登録する
  [引数] pcal -- カロリーリストへのポインタ
         num  -- 登録前のリストの要素数
  [戻り値] 登録後のリストの要素数
*******************************************/
int calregist(CALORIE *pcal, int num)
{
    printf("食品名を入力してください : ");
    scanf("%s", (pcal+num)->name);
    printf("その食品のカロリーを入力してください[kcal/100g] : ");
    scanf("%f", &((pcal+num)->value));
    printf("登録しました。¥n¥n");
    return num+1;
}
```

CALORIE構造体の定義

プロトタイプ宣言

データベースはmain( )関数内で定義（最大500件）

最初の6件は登録済み

```
/*********************************************
  calcalc() カロリーを計算する
  [引数] pcal -- カロリーリストへのポインタ
         num  -- リストの要素数
  [戻り値] カロリー数
*********************************************/
float calcalc(CALORIE *pcal, int num)
{
    char name[40]; /* 入力した食品名 */
    float gram;    /* 入力したグラム数 */
    float totalcal = 0.0; /* 合計カロリー */
    int i;

    printf("--食品名一覧---------------¥n");
    for(i = 0; i < num; i++)
        printf("%s¥t", (pcal+i)->name);
    printf("¥n---------------------------¥n");

    while(1) {
        printf("食品名(endで計算) : ");
        scanf("%s", name);
        if(strcmp(name, "end") == 0)
            break;
        printf("グラム数 : ");
        scanf("%f", &gram);
        for(i = 0; i < num; i++) {
            if(strcmp(name, (pcal+i)->name) == 0) {
                totalcal += (pcal+i)->value * gram / 100.0;
                break;
            }
        }
    }
    return totalcal;
}
```

実行結果

※太字はキーボードから入力した文字

```
カロリー計算ツール
登録は1を、計算は2を、終了は0を入力してください : 1 ↵
食品名を入力してください:いちご ↵
その食品のカロリーを入力してください[kcal/100g] : 36.4 ↵
登録しました。

登録は1を、計算は2を、終了は0を入力してください : 2 ↵
--食品名一覧--------------
米飯  中華麺 そば  うどん 素麺  食パン いちご
--------------------------
食品名(endで計算): そば ↵
グラム数 : 120 ↵
食品名(endで計算) : いちご ↵
グラム数 : 50 ↵
食品名(endで計算) : end ↵
総カロリー : 178.16kcal

登録は1を、計算は2を、終了は0を入力してください : 0 ↵
```

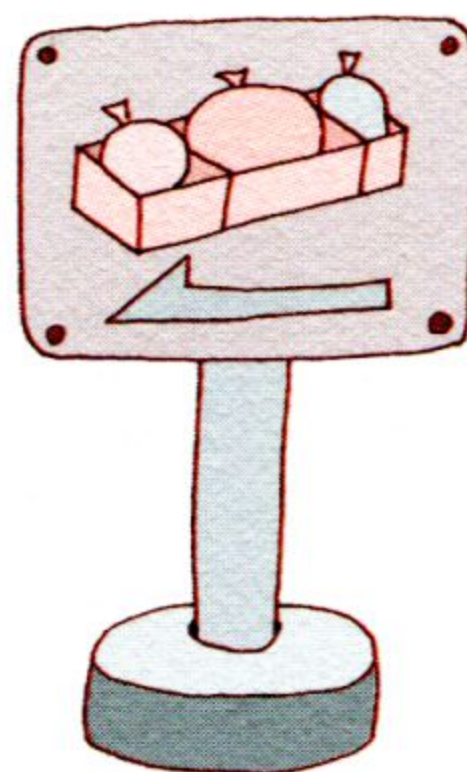
COLUMN コラム

～データをまとめる～

C言語プログラミングも佳境に入ってきました。この本を最初から読み進めてきた人もそろそろまとまったプログラムが書けるようになってきたのではないでしょうか。処理の内容が高度になってくると、処理の単位ごとにいくつかの関数に分けたほうが見やすく、効率がよくなってきます。さらに慣れてくると、グローバル変数をできるだけ使わず、引数を指定するように作った方が関数が汎用的になる、というメリットが感じられるようになると思います。

このようにプログラムの構造化を突き詰めていくと、おのずと関数の引数は増え、その内容も複雑になってきます。引数の数が10個に及んだり、引数の型がポインタのポインタになることだってあるのです。たとえば、次の2つでは、関数を利用する頻度が多いほど、後者の方がシンプルになるでしょう。

```
void getpoint(int *x, int *y, int *z, int *col)
{
        :
}

int x, y, z, col;
        :
getpoint(&x, &y, &z, &col);
        :
```

```
typedef struct _POS3D {
    int x, y, z, col;
} POS3D, *LPPOS3D;

void getpoint(LPPOS3D pos)
{
        :
}

POS3D pos3d;
        :
getpoint(&pos3d);
        :
```

また、引数が多い場合、ひとつひとつを値渡しにすると、その度にデータの移動が発生しますが、参照渡しなら構造体へのポインタ1つ渡せば済むので、速度的にも有利です。ただし、参照渡しでは、関数内での値の変更が呼び出し元に反映されてしまいますので、気をつけるようにしましょう。

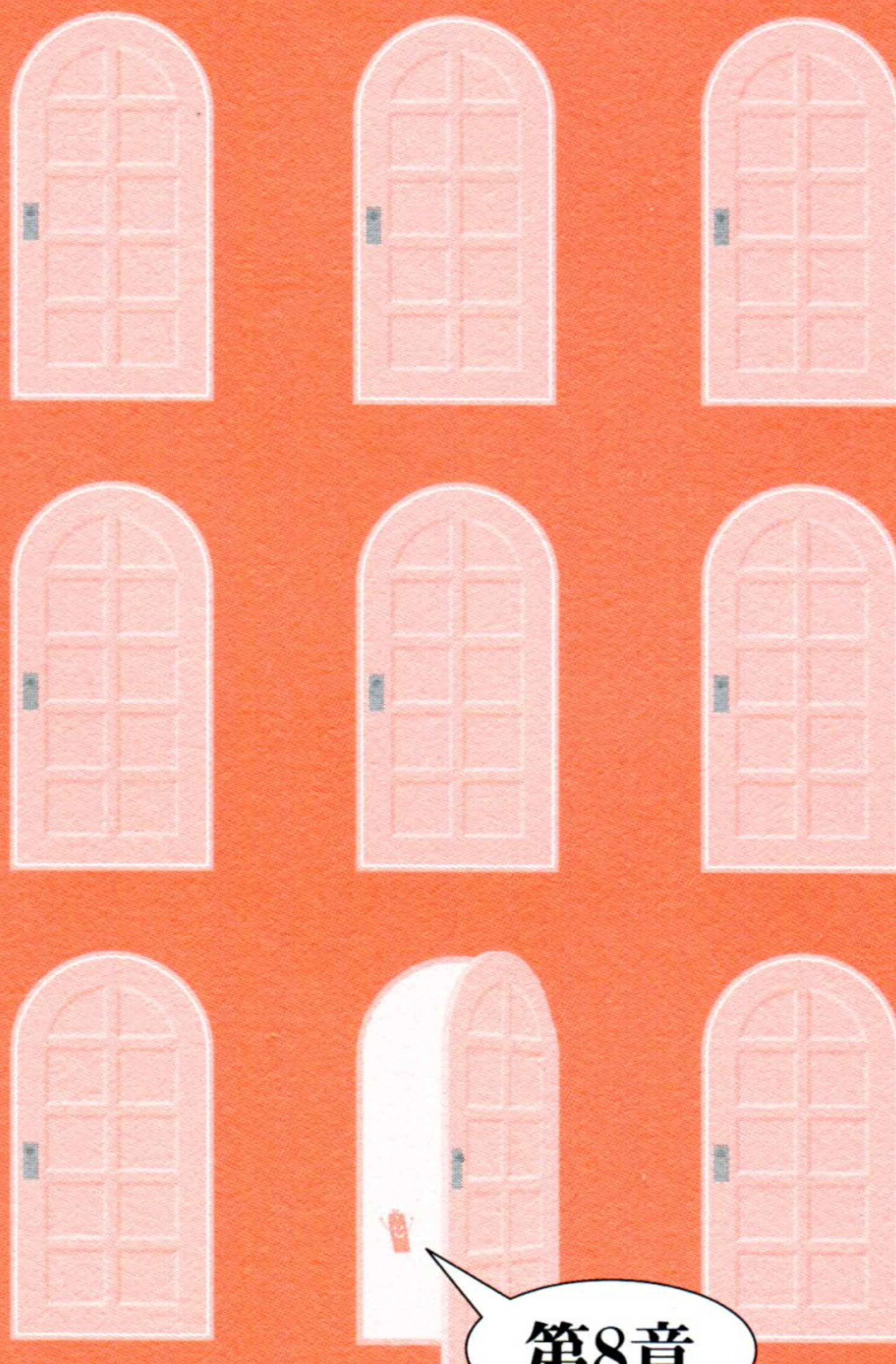
構造体を1つの「かたまり（オブジェクト）」と考え、まとめてやりとりしようという考え方は、やがてC++言語の「クラス」の考え方につながっていきます。

プログラムの構成

第8章

第8章は

ここがkey!

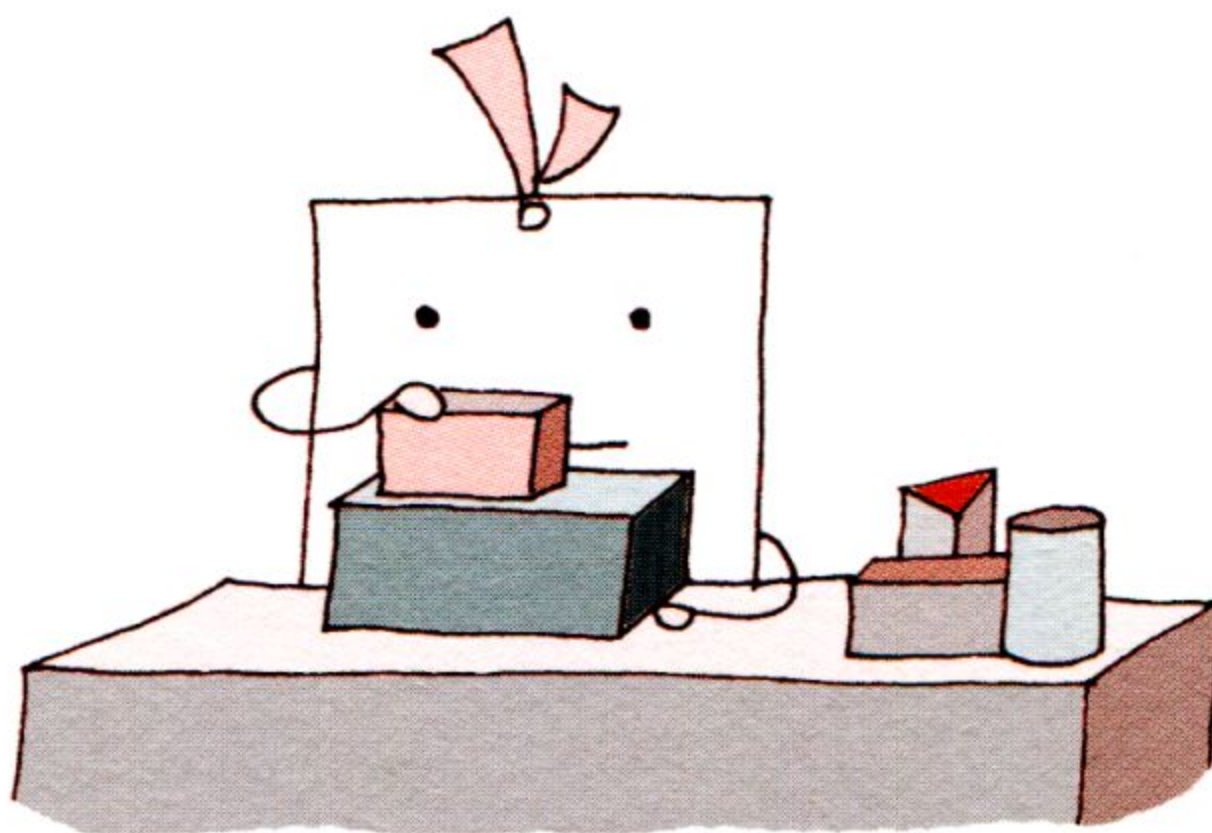
Topics

コーディングだけではない、腕の見せどころ

たとえば、printf()関数を使うには、ソースファイルの冒頭で、「#include <stdio.h>」と記述する必要がありました。この行にはどのような意味があるのでしょうか？

実は、この記述は、「stdio.hというファイルを、ソースファイルの中に組み込みなさい」という命令なのです。拡張子「.h」のファイルを**ヘッダファイル**といいます。ヘッダファイルには、主に宣言や定義が書いてあります。拡張子「.c（.cppなどのときもあります)」のソースファイルでは、**#include**を使って、ヘッダファイルを**インクルード**して、その機能を取り込みます。

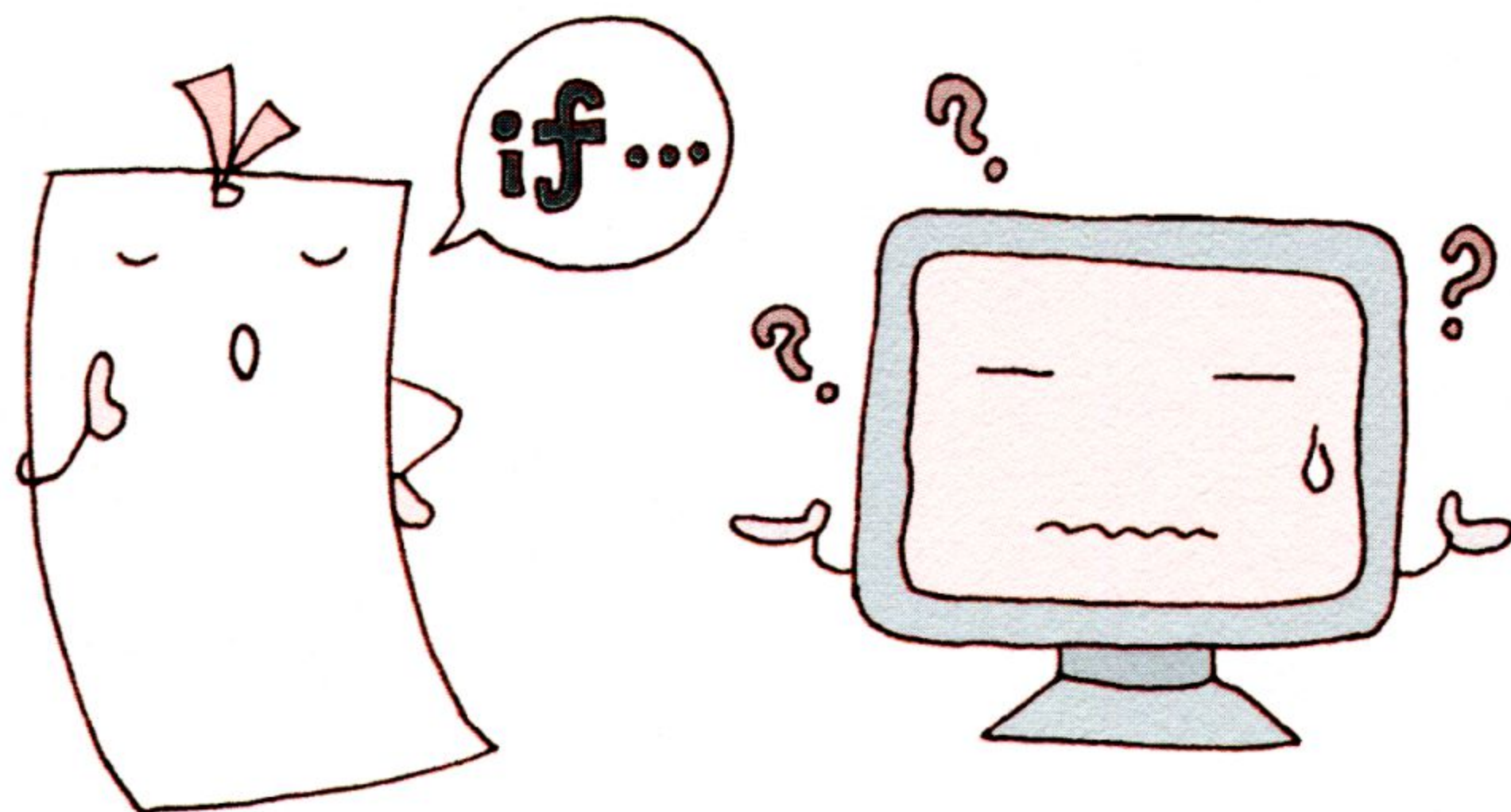
このように、C言語のプログラムは複数のファイルから構成される場合がほとんどです。大きなプログラムは、何十、何百ものヘッダファイルやソースファイルから構成されます。

コーディングの技術だけでなく、これらのファイルをどう組み立てるかといった設計技術も、プログラマの腕の見せどころです。プログラムの機能や規模に応じて、適切な設計を考えるのは、そう簡単なことではありません。なによりも、経験がものをいうところかもしれません。この章では、手はじめにプログラムを構成するファイルをどのように組み立てたらよいか、という基本的なことを押さえておくことにしましょう。

Topics

プログラマとコンピュータ ～実行ファイルができるまで

プログラマとコンピュータの関係にも目を向けてみます。C言語は、コンピュータに対する命令文ですね。しかし、第2章の冒頭でふれたとおり、0と1（2進数）の世界で動くコンピュータは、ifやswitchといった言葉を理解することができません。それでは、どのようにして、プログラマの命令がコンピュータに伝わり、プログラムが動作しているのでしょうか？

この謎を解くために、プログラムの実行ファイルができるまでの過程を見ていきたいと思います。おおまかにいってプログラムのソースファイルは、**コンパイル**、**リンク**、という2つの工程を経て、実行ファイルになります。C言語のソースプログラムをコンパイルすると、それらはコンピュータに理解できる「機械語」に翻訳され、オブジェクトファイルというファイルになります。リンク処理は、オブジェクトファイルなどを、1つのファイルにまとめて、実行ファイルを作ります。

なお、コンパイルの一番最初の段階では、「#」からはじまるキーワードで指定した命令が実行されます（#includeによるファイルの組み込みも、このときに実行されます）。この命令のことを、**マクロ**といいます。章の後半では、マクロの便利な活用方法と、注意すべき点について説明します。

ヘッダファイル

拡張子「.h」のヘッダファイルの内容や使い方について、詳しく見ていきましょう。

ヘッダファイルの内容

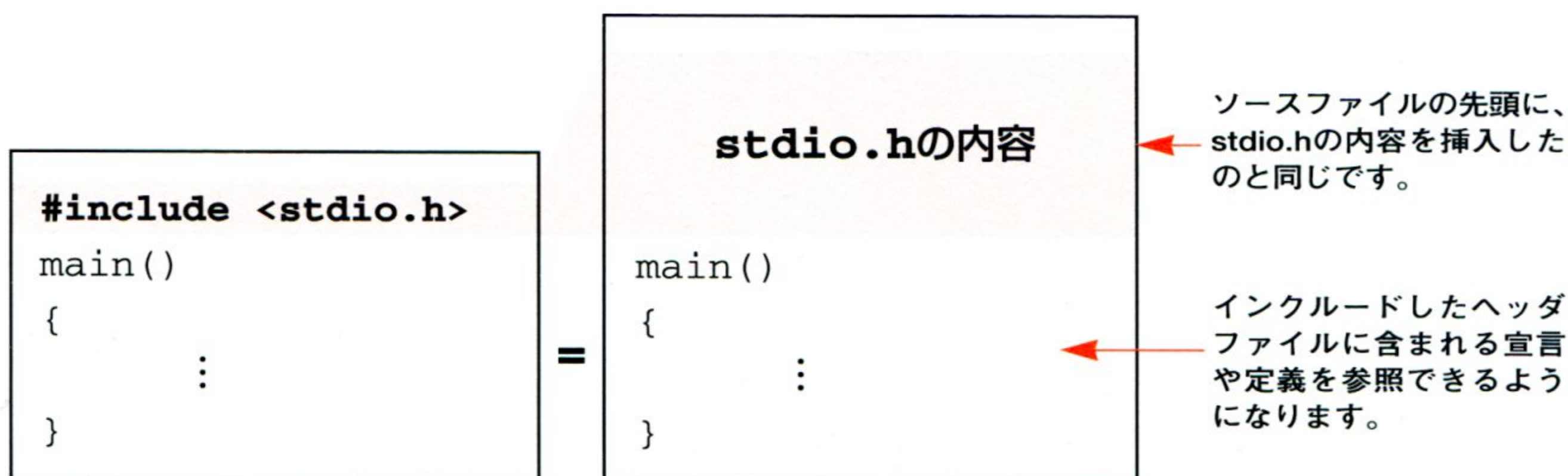
stdio.hなど、拡張子が「.h」のファイルのことを、ヘッダファイルといいます。ヘッダファイルは、プロトタイプ宣言、構造体や定数の定義などを含むテキストファイルで、これをソースファイルの中に組み込む（インクルードする）と、それらの宣言や定義を利用できるようになります。

```
#include <stdio.h>
main()
{
        :
}
```

＝

```
stdio.hの内容
main()
{
        :
}
```

ソースファイルの先頭に、stdio.hの内容を挿入したのと同じです。

インクルードしたヘッダファイルに含まれる宣言や定義を参照できるようになります。

インクルードの書式は、C言語に標準で用意されているヘッダファイルと、自分で作ったヘッダファイルで異なります。

書式1

```
#include <ファイル名>
```

標準関数のヘッダファイルをインクルード

書式2

```
#include "ファイル名"
```

呼び出し元と同じ場所の自作ヘッダファイルをインクルード

C言語が用意しているヘッダファイルには次のようなものがあり、処理の種類ごとに別々のファイルに分かれています。ソースファイルの中で標準ライブラリ関数を呼び出すには、適切なヘッダファイルをインクルードする必要があります。

ヘッダファイル	処理の種類
stdio.h	入出力
string.h	文字列処理
time.h	時間処理
math.h	数学処理

ヘッダファイルの場所を調べれば、その内容を見ることもできますが、変更してはいけません。

C ヘッダファイルを作る

プログラマが自作のヘッダファイルを用意することもできます。その場合、独自に定義した関数や構造体、マクロの宣言や定義をヘッダファイルに記述します。代入式など、具体的な処理を書くべきではありません。それら宣言や定義を利用する場合は、そのヘッダファイルをインクルードする必要があります。

mysource.c

```
#include <stdio.h>
void myfunc(void);
main()
{
        ⋮
}
void myfunc()
{
        ⋮
}
```

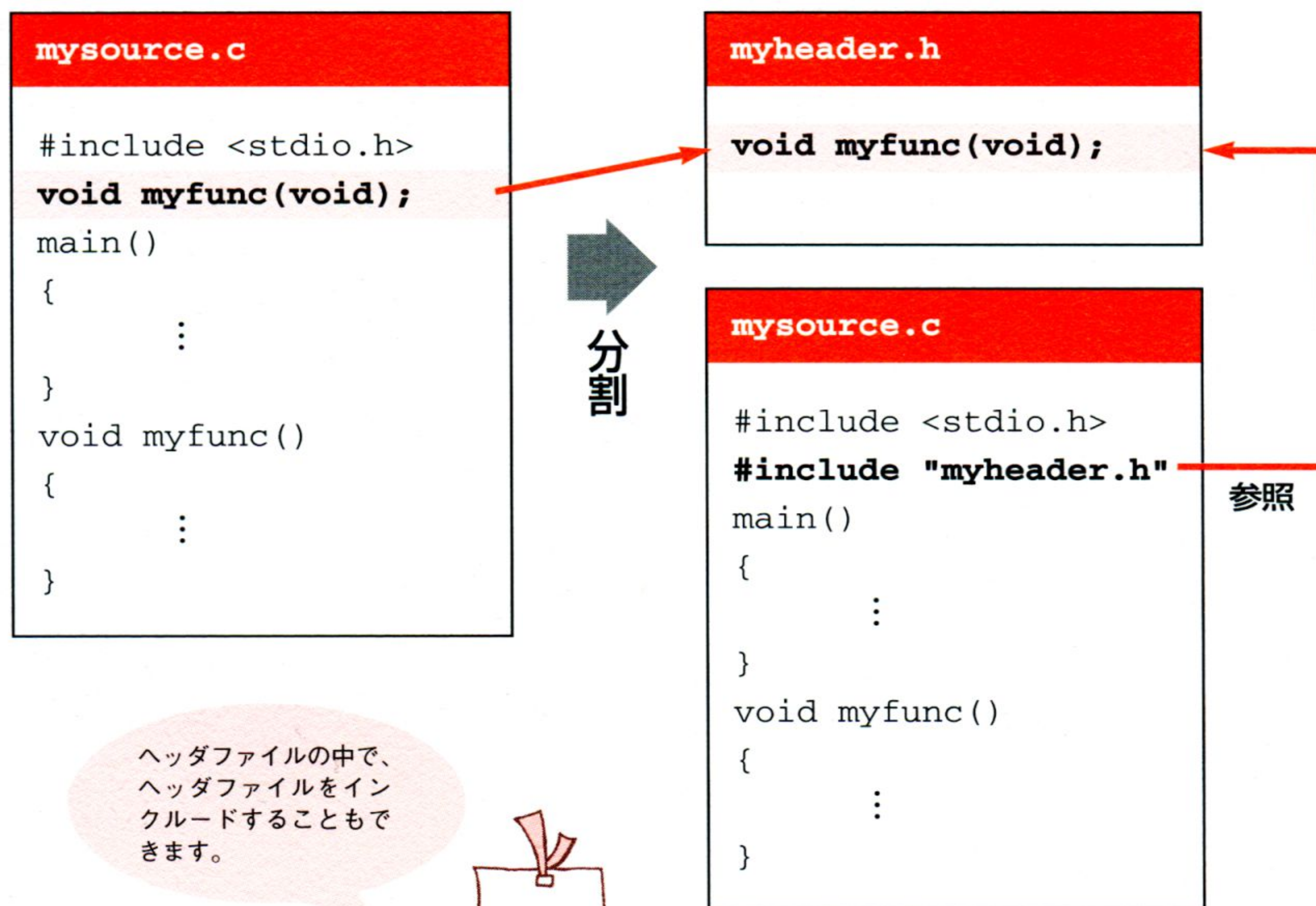
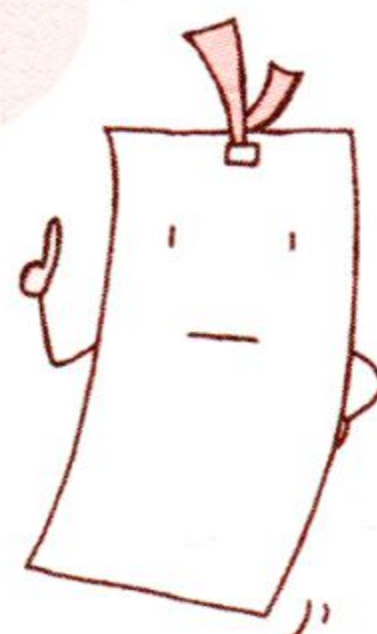
分割

myheader.h

```
void myfunc(void);
```

mysource.c

```
#include <stdio.h>
#include "myheader.h"
main()
{
        ⋮
}
void myfunc()
{
        ⋮
}
```

参照

ヘッダファイルの中で、ヘッダファイルをインクルードすることもできます。

コンパイルとリンク

C言語のプログラミングの実行ファイルができるまでの過程を紹介します。

実行ファイルができるまで

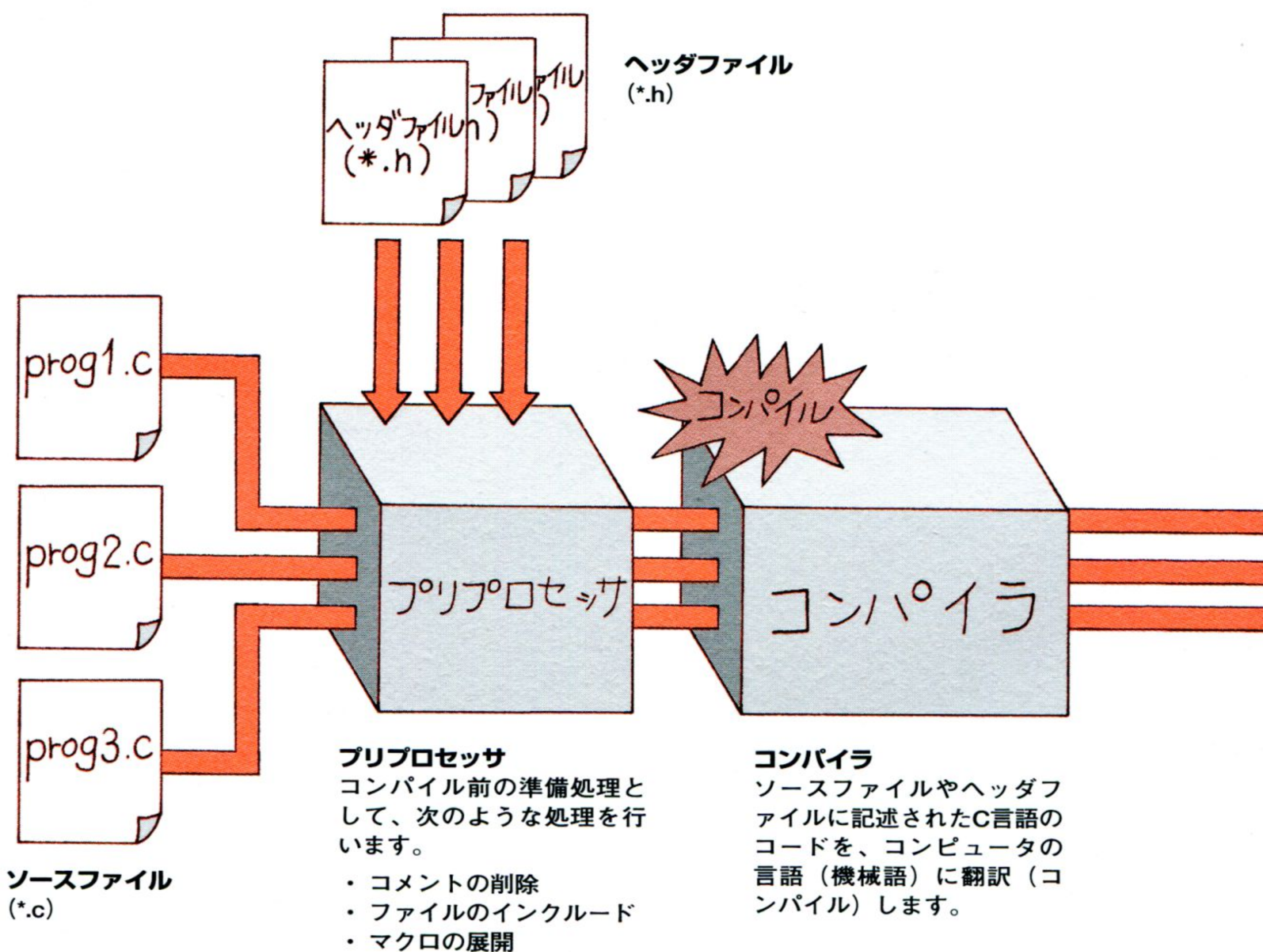
プログラムの実行ファイルは、C言語のソースコードをコンパイル、リンクすることにより、できあがります。コンパイルとリンクを合わせて**ビルド**または**メイク**といいます。

ヘッダファイル
(*.h)

ソースファイル
(*.c)

プリプロセッサ
コンパイル前の準備処理として、次のような処理を行います。

- コメントの削除
- ファイルのインクルード
- マクロの展開

コンパイラ
ソースファイルやヘッダファイルに記述されたC言語のコードを、コンピュータの言語（機械語）に翻訳（コンパイル）します。

下の手順をひとつひとつ実行することもできますが、コンパイラ製品にはたいてい、**メイクプログラム**と呼ばれる自動処理ツールが用意されています。メイクプログラムは、**メイクファイル**というテキストファイルを元に、最小の手順で自動的に実行ファイルを作ります（Visual C++ではプロジェクトでこれらを管理しています）。

ライブラリファイル
機械語で書かれた関数やデータの集まりです。標準ライブラリ関数の実体も、ライブラリファイルの中にあります。

ライブラリファイル
(*.lib,*.a)

必要な部分だけ（たとえばprintf( )関数の実装だけ）を抜粋してリンクします。

prog1.obj
prog2.obj
prog3.obj

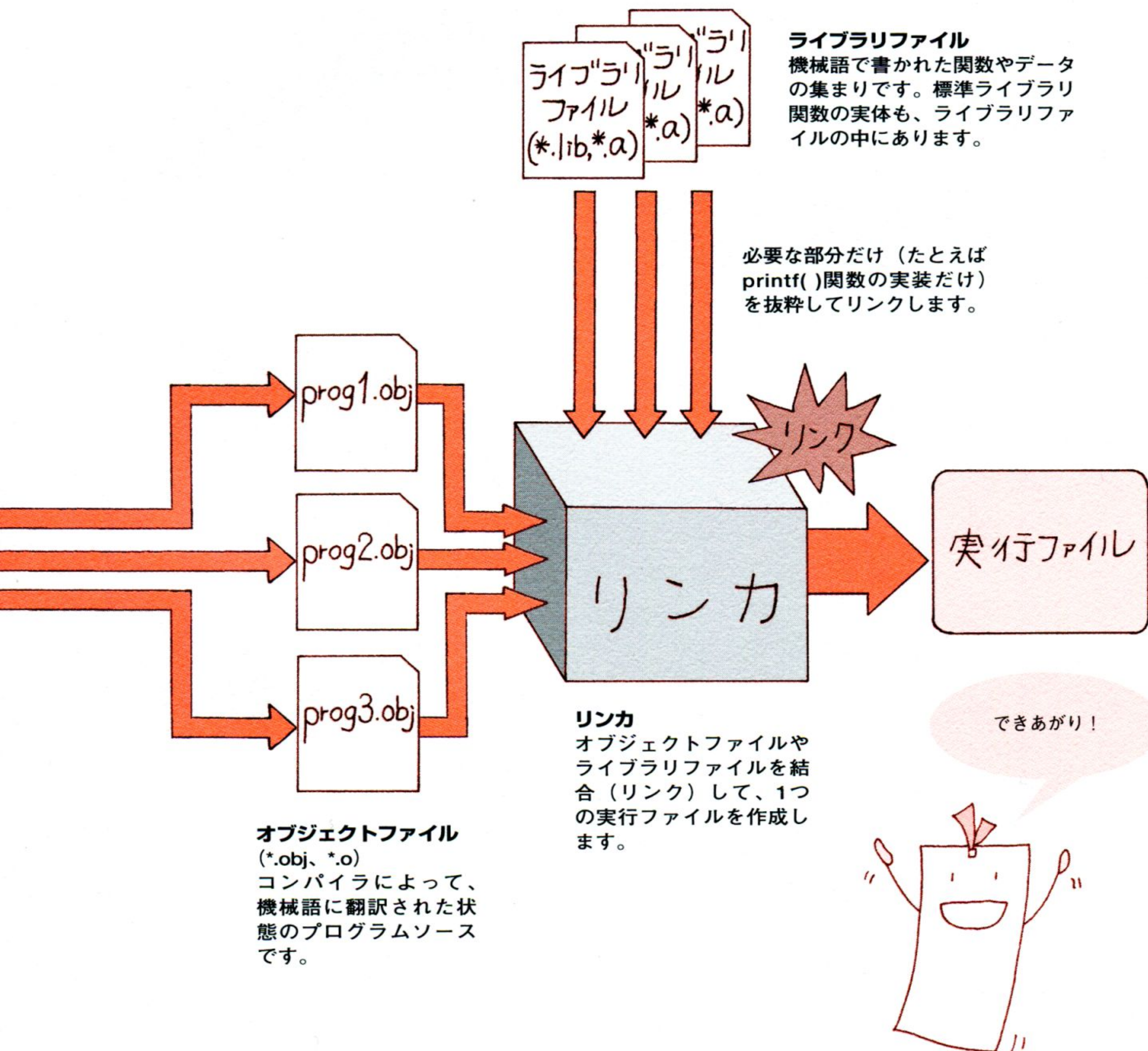
リンク

リンカ

実行ファイル

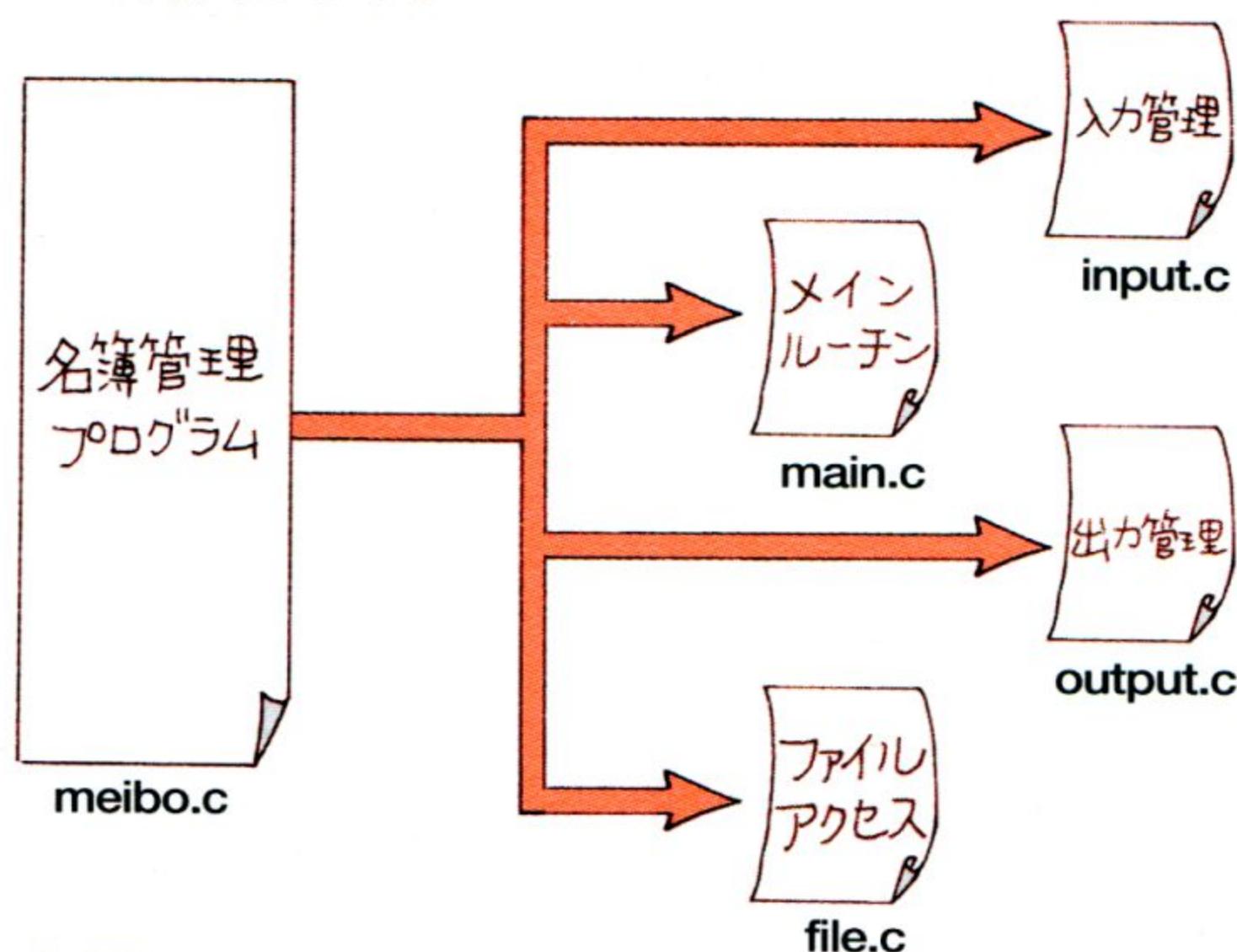
リンカ
オブジェクトファイルやライブラリファイルを結合（リンク）して、1つの実行ファイルを作成します。

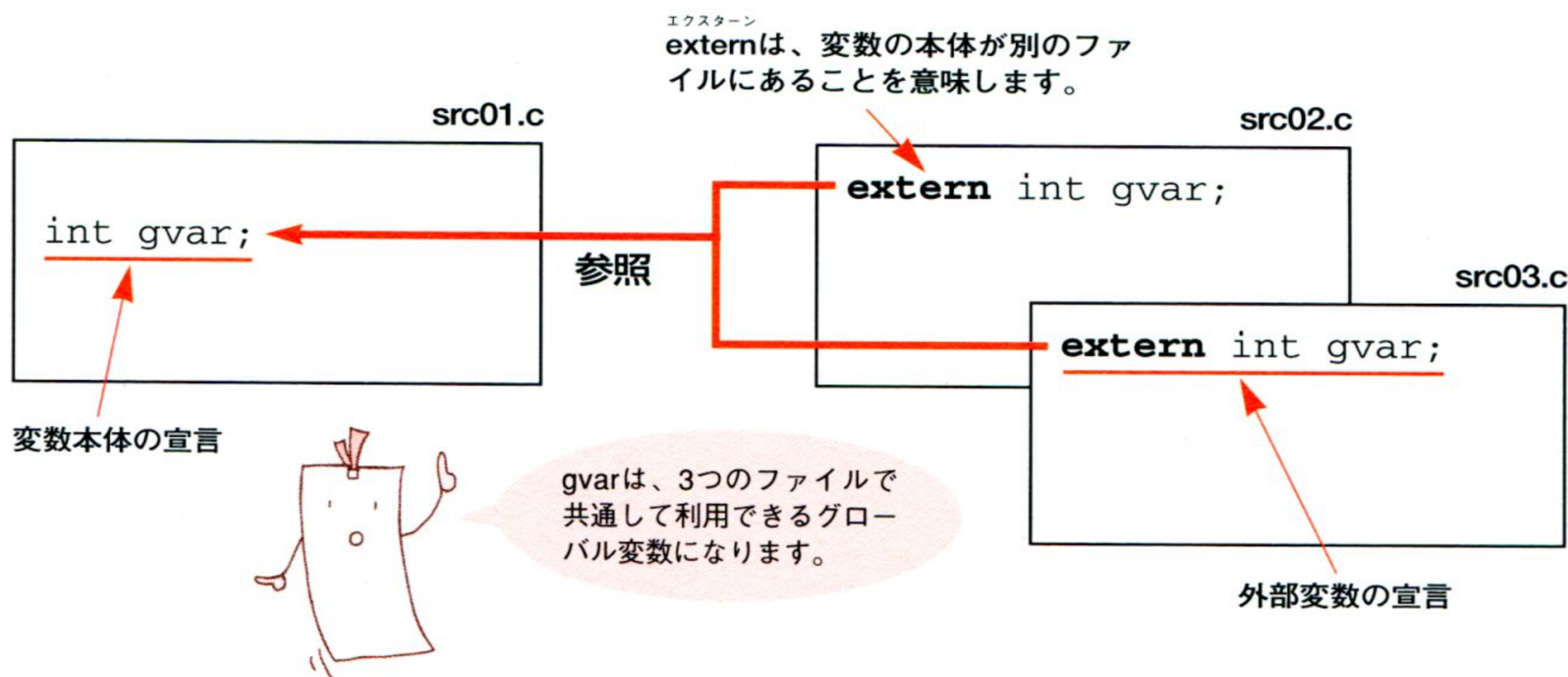
できあがり！

オブジェクトファイル
(*.obj、*.o)
コンパイラによって、機械語に翻訳された状態のプログラムソースです。

ファイルの組み立て

ソースファイル(*.c)とヘッダファイル(*.h)を、どのように用意したらよいか、考えてみましょう。

ソースファイルの分割

ソース管理や読みやすさを考えると、ひとつのソースファイルに大量のプログラムコードを記述するのは、好ましくありません。そのような場合は、プログラムを構成する「機能」ごとにソースファイルを分割します。

外部変数宣言

ソースファイルを分割した場合、異なるソースファイル間の連携が問題になります。外部変数を利用すると、複数のファイルから共通の変数を参照できます。実際の変数宣言も、外部変数の宣言も、関数の外に記述します。

C ヘッダファイルの利用

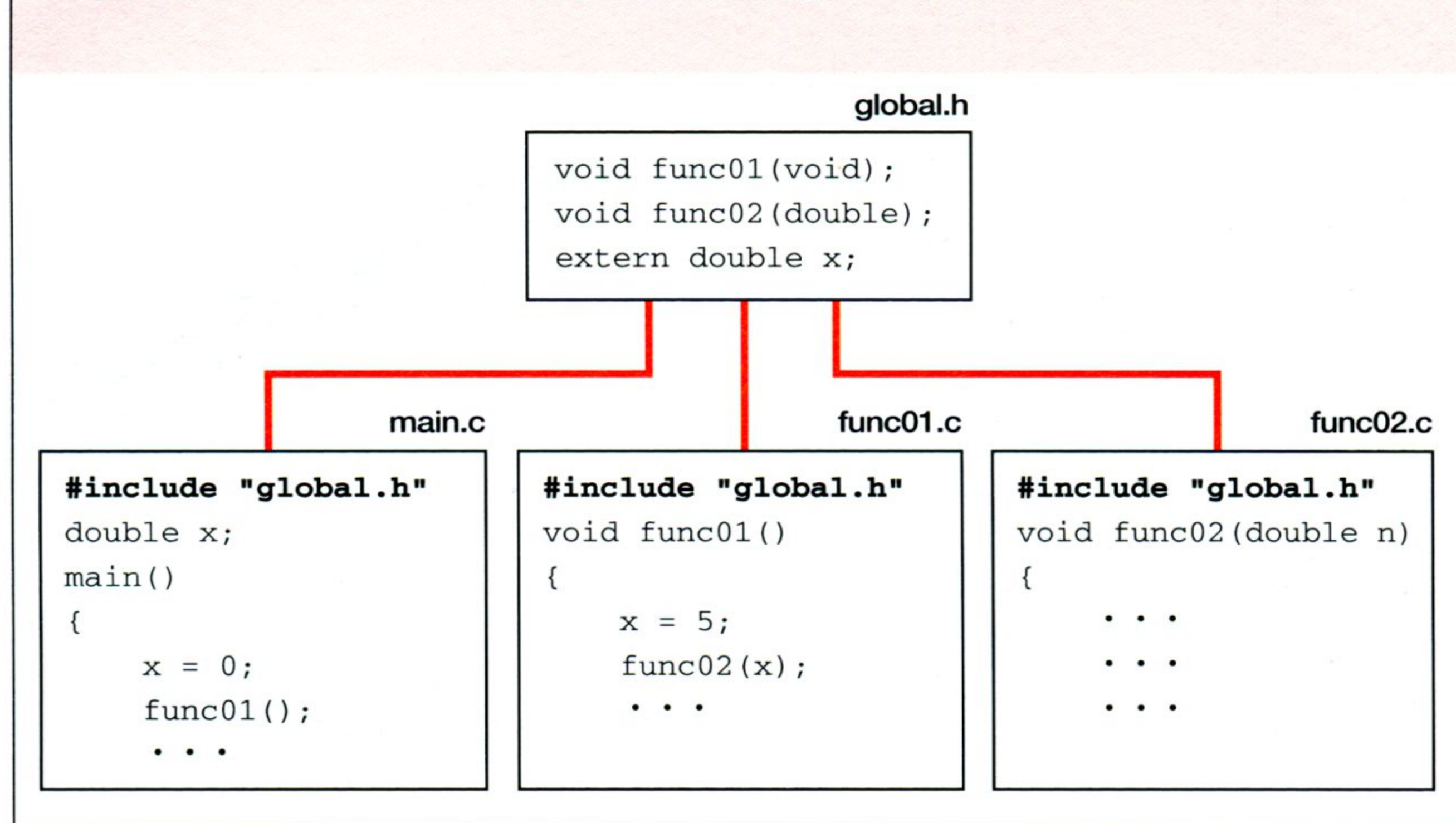
ヘッダファイルを利用することにより、さらに効率よく、プロジェクトを管理することができます。構成の一例を考えてみましょう。

各ソースファイルで共通して利用する宣言や定義を、1つのヘッダファイルにまとめる。

global.h
```
void func01(void);
void func02(double);
extern double x;
```

main.c
```
#include "global.h"
double x;
main()
{
    x = 0;
    func01();
    ・・・
```

func01.c
```
#include "global.h"
void func01()
{
    x = 5;
    func02(x);
    ・・・
```

func02.c
```
#include "global.h"
void func02(double n)
{
    ・・・
    ・・・
    ・・・
```

ソースファイルと対になるヘッダファイルを作成し、それぞれに、他のファイルに公開する宣言や定義を記述する。

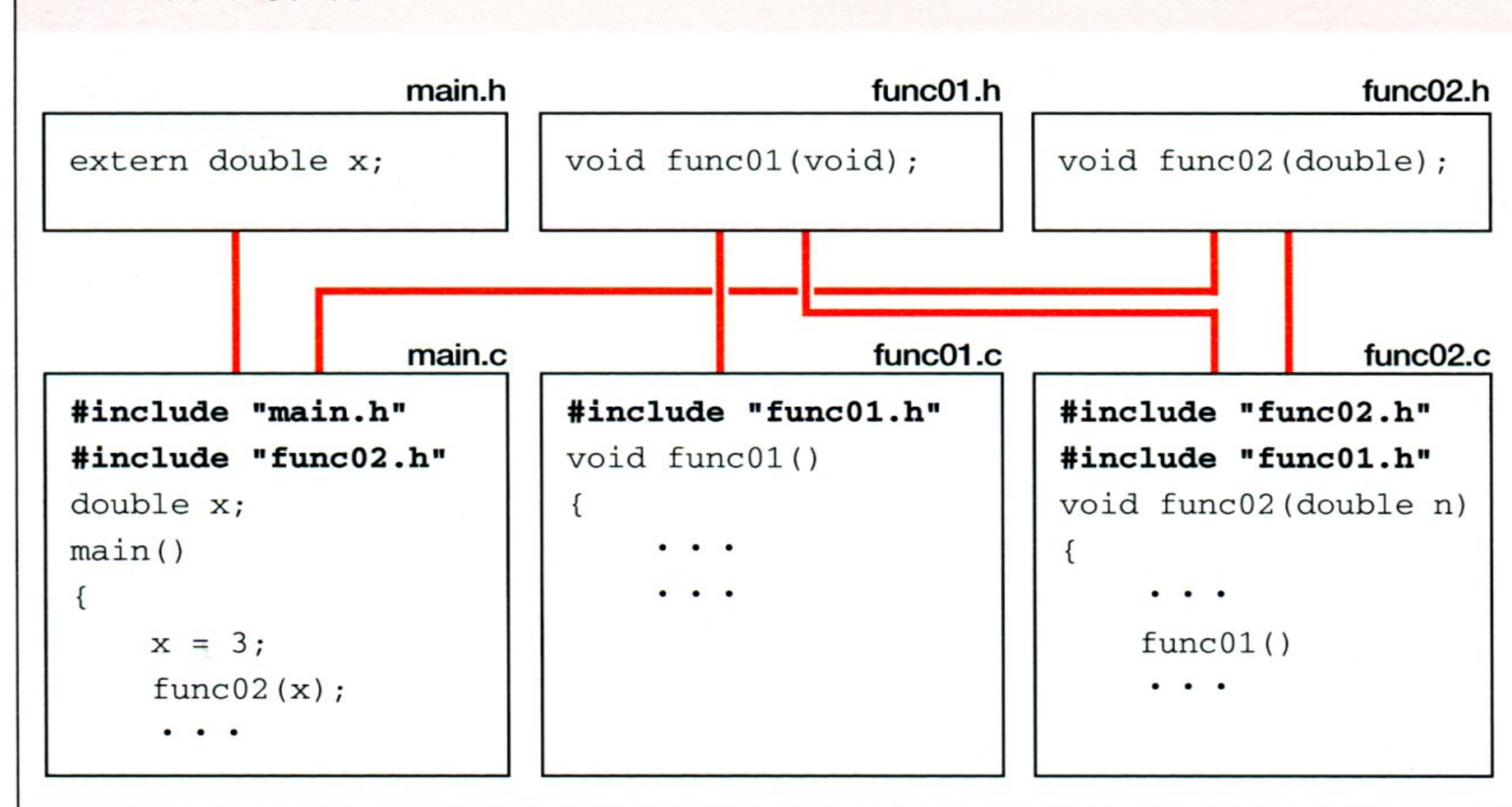
main.h
```
extern double x;
```

func01.h
```
void func01(void);
```

func02.h
```
void func02(double);
```

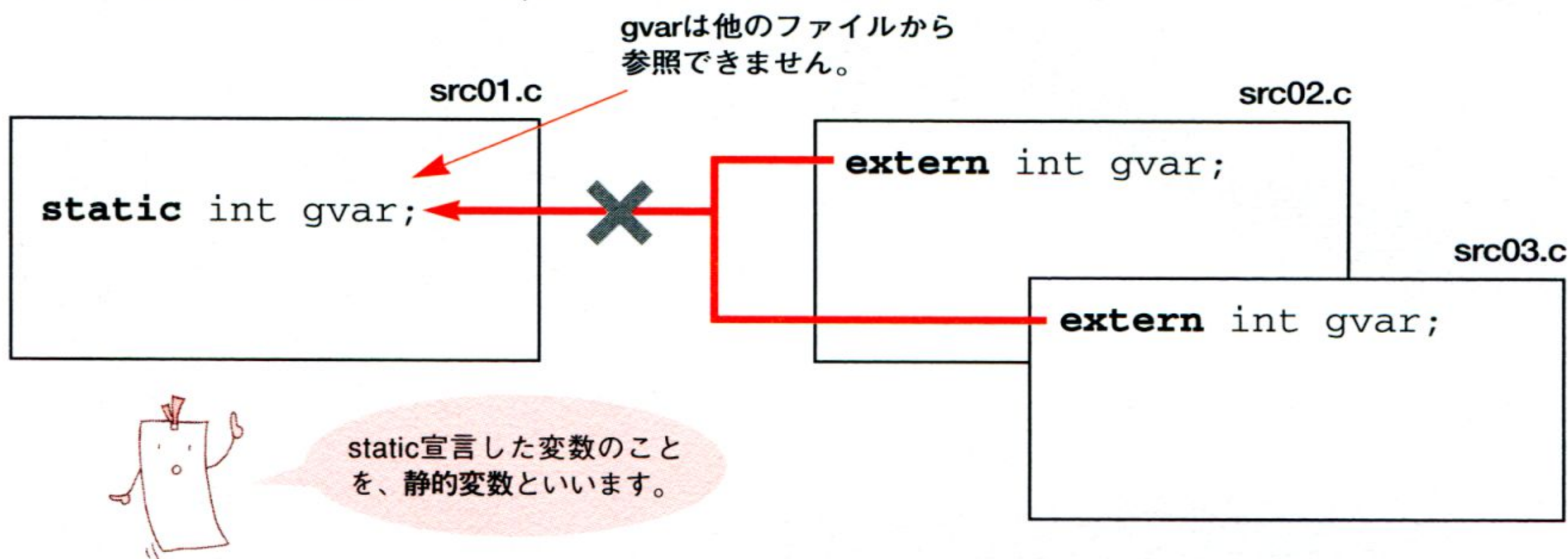
main.c
```
#include "main.h"
#include "func02.h"
double x;
main()
{
    x = 3;
    func02(x);
    ・・・
```

func01.c
```
#include "func01.h"
void func01()
{
    ・・・
    ・・・
```

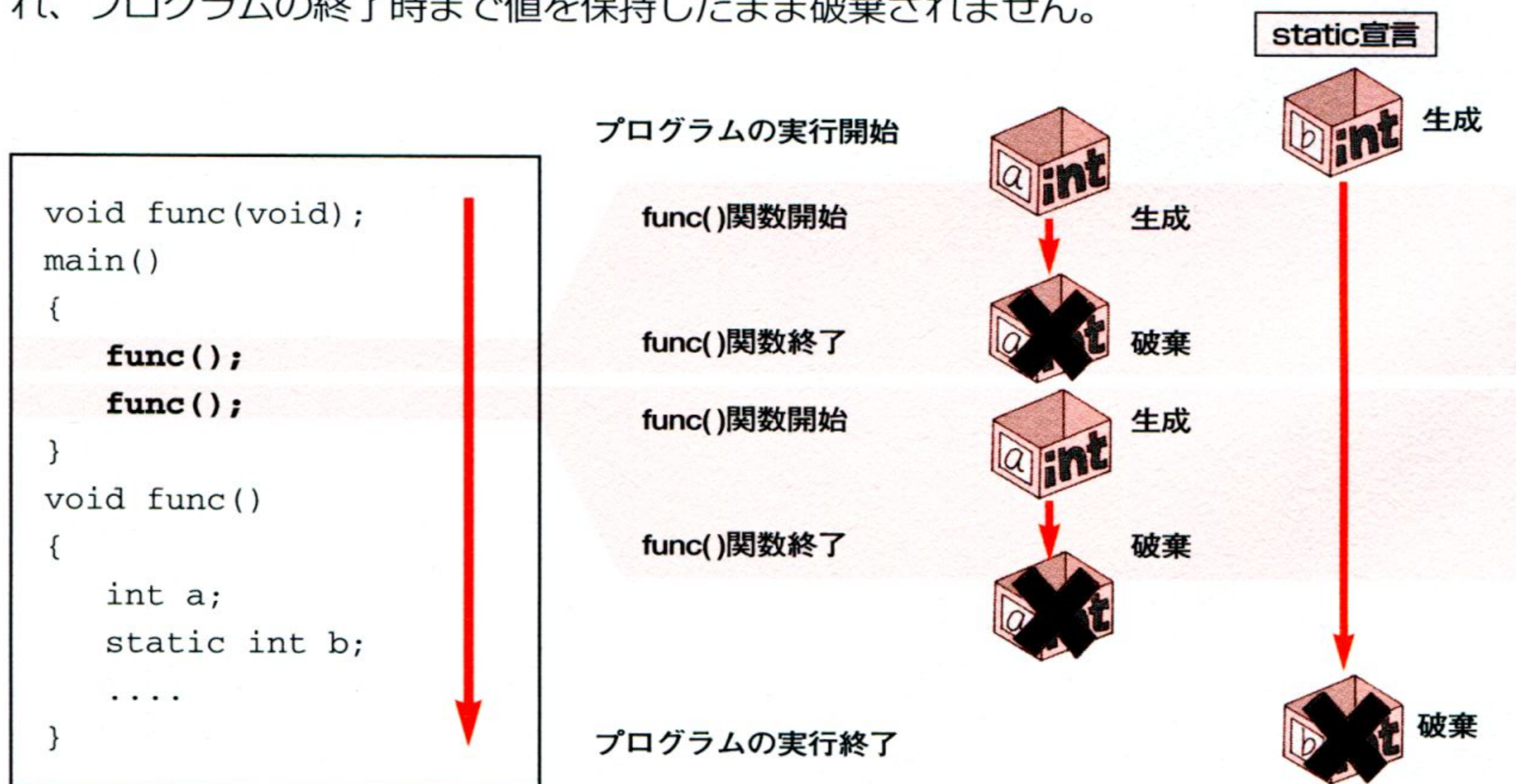
func02.c
```
#include "func02.h"
#include "func01.h"
void func02(double n)
{
    ・・・
    func01()
    ・・・
```

いろいろな宣言

変数をstatic（スタティック）宣言すると、外部からの参照を制限したり、変数の有効期限を変更することができます。

static宣言したグローバル変数

staticをつけてグローバル変数を宣言すると、変数の有効範囲は、その宣言したファイルの中に限られます。外部変数を宣言して、他のファイルから参照することはできません（リンク時にエラーとなります）。

static宣言したローカル変数

通常、ローカル変数は関数の実行を開始するときに作られ、関数が実行を終了する際に破棄されます。しかし、staticで宣言したローカル変数はプログラムの実行開始時に作られ、プログラムの終了時まで値を保持したまま破棄されません。

```
void func(void);
main()
{
    func();
    func();
}
void func()
{
    int a;
    static int b;
    ....
}
```

const宣言

（コンスト）

次のように変数をconst宣言すると、変数の値を変更できなくなります。つまり、const宣言した変数は、定数として扱うことができます。

```
const int x = 10;
```

整数値10を意味する定数xを、宣言します。

関数の引数がconst宣言されている場合、その変数の値を関数の中で変更しないことが保証されています。

標準ライブラリ関数strcatのプロトタイプ宣言

```
char *strcat(char *str1, const char *str2);
```

例

```
#include <stdio.h>

void increment(void);

main()
{
    int i;
    for(i = 0; i < 3; i++)
        increment();
}

void increment()
{
    int a = 0;
    static int b = 0;
    a++; b++;
    printf("a:%d, b:%d¥n", a, b);
}
```

はじめて呼びだされたときに初期化されます。

実行結果

```
a:1, b:1
a:1, b:2
a:1, b:3
```

static宣言したbだけ、関数呼び出しのたびに値が増加していきます。

マクロ(1)

マクロを利用することによって、コンパイルに関係するさまざまな処理を指定することができます。

マクロとは？

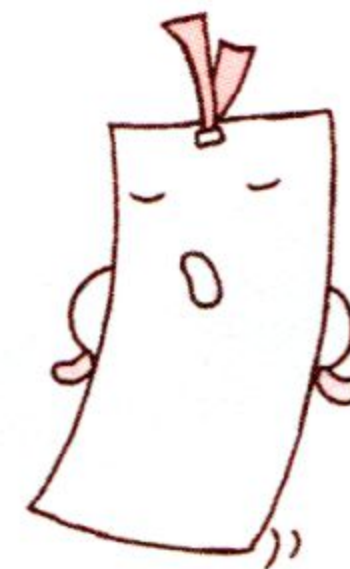
C言語では、#からはじまる1行分のことをマクロと呼んでいます。マクロは、今まで学んできた文法の枠を超えた拡張的な機能で、コンパイル前にプリプロセッサが処理します。この章ですでに説明した#includeも、マクロのひとつです。

置換 － #define（デファイン）

#defineは、文字列を置換するマクロです。
次のように書くと、プログラム中にあるLOOPNUMという記述を3に置換します。

スペースで区切ります。

```
#define LOOPNUM 3
```

セミコロンはつけません（セミコロンも置換の対象になります）。

置換される側は、他の変数などと区別するため、大文字にします。

また、次のように書くと、「DEBUG_MODEが定義されている」という事実のみを表します。

```
#define DEBUG_MODE
```

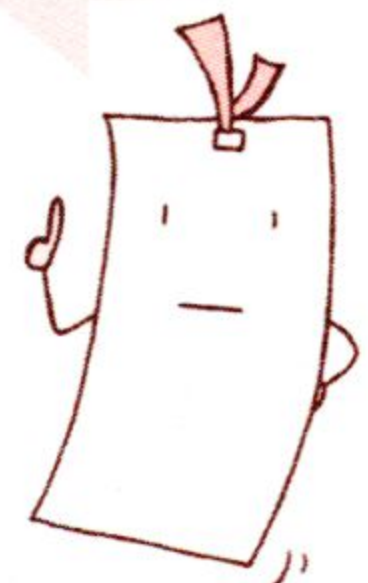
例

```
#include <stdio.h>

#define LOOPNUM 3

main()
{
    int i;
    for(i = 0; i < LOOPNUM; i++)
        printf("LoopCount:%d¥n", i+1);
}
```

この行以降に出てくるLOOPNUMという記述を3に置き換えます。

3に置き換わります。

実行結果

```
LoopCount:1
LoopCount:2
LoopCount:3
```

条件に応じたコンパイル指示 − #if、#ifdef、#ifndef

条件に応じて、必要な部分だけ抜き出してコンパイルしたいときがあります。そのようなときは、次の書式でマクロを定義します。

```
#if 条件
    指定範囲
#endif
```

条件が真のとき、指定範囲をコンパイルの対象領域に含める。

```
#ifdef 識別子
    指定範囲
#endif
```

識別子が**定義されている**とき、指定範囲をコンパイルの対象領域に含める。

```
#ifndef 識別子
    指定範囲
#endif
```

識別子が**定義されていない**とき、指定範囲をコンパイルの対象領域に含める。

if文と同じように、複数の条件を判断することもできます。

```
#ifdef 識別子
    指定範囲A
#elsif 条件B
    指定範囲B
#else
    指定範囲C
#endif
```

識別子が定義されている
→ 指定範囲Aをコンパイル

条件Bが成立
→ 指定範囲Bをコンパイル

どれも成立しない
→ 指定範囲Cをコンパイル

コンパイル領域に含められる指定範囲は、どれか1つです。

ヘッダファイルの重複インクルードを防ぐ

いろいろなファイルで#includeを使うと、結果的に同じヘッダファイルを2度以上インクルードしてしまうことがあります。しかし、そうすると宣言が重複してエラーになってしまいます。
ヘッダファイルに次のようなマクロを付加して、重複インクルードを防ぐことができます。

myheader.h

```
#ifndef _MYHEADER_
#define _MYHEADER_
    void MyFunc();
    extern int x;
#endif
```

最初のインクルードで_MYHEADER_を定義します。

2回目以降は、すでに_MYHEADER_が定義されているので、この内容はインクルードできません。

マクロ(2)

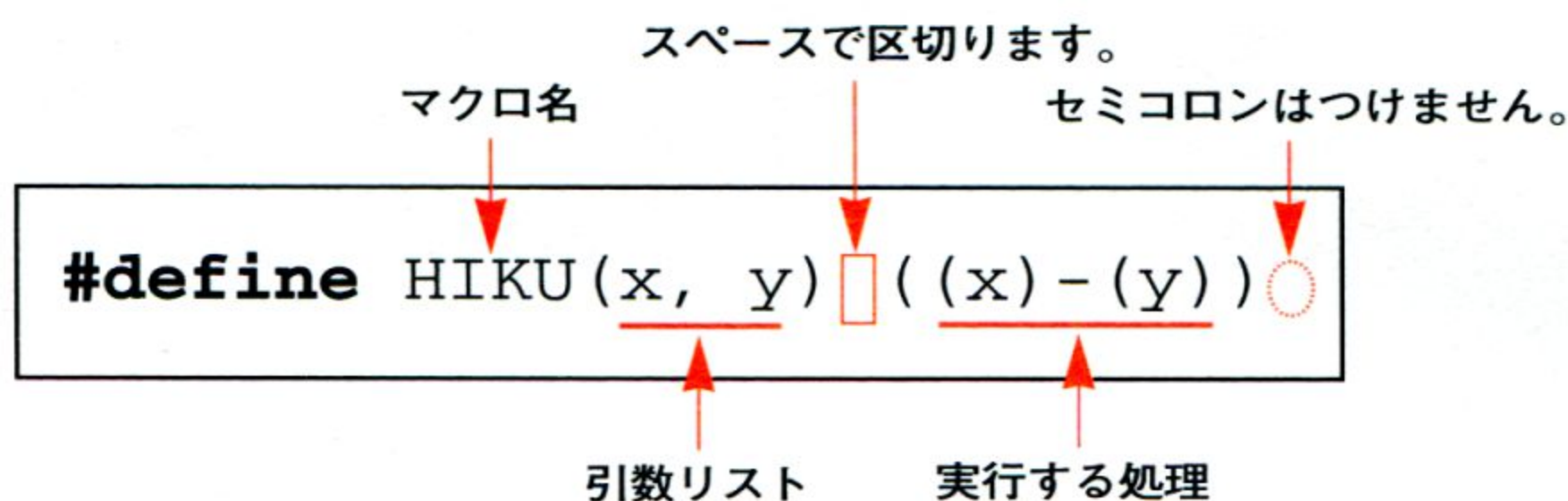
演算子の優先順位など注意点に十分配慮して、引数つきマクロを利用しましょう。

引数つきマクロ

#define文を使うと、引数を持ち、関数のように動作するマクロを定義することができます。次の例ではx-yの値を求めるマクロHIKUを定義します。

スペースで区切ります。
マクロ名
セミコロンはつけません。

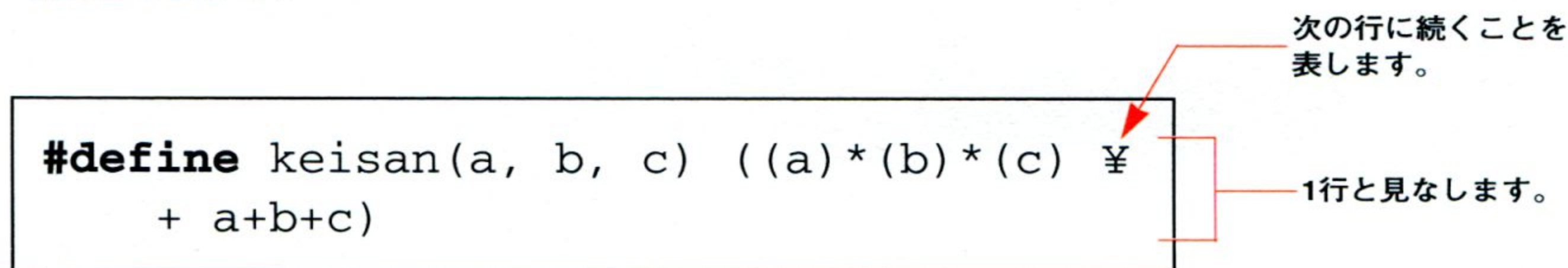
```
#define HIKU(x, y) ((x)-(y))
```

引数リスト
実行する処理

引数を対応づけて置換します。

152ページで述べたように、#行の有効範囲は1行ですが、¥記号を使うと複数行にわたって記述できます。

次の行に続くことを表します。

```
#define keisan(a, b, c) ((a)*(b)*(c) ¥
    + a+b+c)
```

1行と見なします。

例

引数つきマクロの定義

```
#include <stdio.h>
#define HIKU(x, y) ((x)-(y))

main()
{
    printf("マクロの実行結果：%d¥n", HIKU(5, 3));
}
```

引数に指定した5と3をセットして置換します。

```
    printf("マクロの実行結果：%d¥n", ((5)-(3)));
```

実行結果

```
マクロの実行結果：2
```

C 引数つきマクロ使用時の注意

演算子の優先順位に配慮して、実行する処理全体とその中の引数はカッコで囲みます。

○
```
#define HIKU(x, y) ((x)-(y))
      ⋮
HIKU(5+2, 1+3)
```
置換

((5+2)-(1+3))=3

×
```
#define HIKU(x, y) (x-y)
      ⋮
HIKU(5+2, 1+3)
```
置換

(5+2-1+3)=9

演算子の優先順位により、意図した結果になりません。

マクロ名と（　）の間にスペースを入れてしまうと、区切りが正しく認識されません。

○
```
#define HIKU(x, y) ((x)-(y))
      ⋮
HIKU(5, 3)
```
置換

((5)-(3))

×
```
#define HIKU (x, y) ((x)-(y))
      ⋮
HIKU(5, 3)
```
スペース

置換

(5,3) ((5)-(3))

例
```
#include <stdio.h>
#define JIJYO(x) ((x)*(x))

main()
{
    int i = 1;
    while(i <= 5){
        printf("結果：%d¥n", JIJYO(i++));
    }
}
```
引数の2乗を計算するマクロを定義。

i++が2回ずつ呼び出されてしまうので、1～5の2乗を順に計算することにはなりません。

実行結果
```
結果：1
結果：9
結果：25
```

サンプルプログラム

■カロリー計算プログラム〈複数ファイル版〉

第7章で紹介したカロリー計算プログラムを3つのファイルに分けると次のようになります。callib.hをインクルードすれば、calorie.c以外でも、callib.hの定義や宣言とcallib.cの処理を利用できます。

ソースコード：**calorie.c**

```
#include <stdio.h>
#include "callib.h"

int main()
{
    CALORIE cal[500] = {
        {"米飯",   150.0},
        {"中華麺",  57.1},
        {"そば",   133.3},
        {"うどん", 100.0},
        {"素麺",   133.3},
        {"食パン", 250.0}
    };
    int cal_num = 6;
    int mode = 0;

    printf("カロリー計算ツール¥n");
    while(1) {
        printf("登録は1を、計算は2を、終了は0を入力してください : ");
        scanf("%d", &mode);
        if(mode == 0)
            break;
        else if(mode == 1)
            cal_num = calregist(cal, cal_num);
        else if(mode == 2)
            printf("総カロリー:%6.2fkcal¥n¥n", calcalc(cal, cal_num));
    }
    return 0;
}
```

自作のヘッダファイルをインクルード

別ファイルで定義した宣言や関数が利用可能

ソースコード：**callib.h**

```
#ifndef _CALLIB_H_
#define _CALLIB_H_

typedef struct _CALORIE {
    char name[40];
    float value;
} CALORIE;

int calregist(CALORIE *, int);
float calcalc(CALORIE *, int);

#endif
```

重複インクルードを避けるためのマクロ

ソースコード：**callib.c**

```
#include <stdio.h>
#include <string.h>
#include "callib.h"          ← 自作のヘッダファイルをインクルード

/*******************************************
  calregist() カロリーリストへ登録する
  [引数] pcal -- カロリーリストへのポインタ
         num  -- 登録前のリストの要素数
  [戻り値] 登録後のリストの要素数
*******************************************/
int calregist(CALORIE *pcal, int num)
{
    printf("食品名を入力してください : ");
    scanf("%s", (pcal+num)->name);
    printf("その食品のカロリーを入力してください[kcal/100g] : ");
    scanf("%f", &((pcal+num)->value));
    printf("登録しました。¥n¥n");
    return num+1;
}

/*******************************************
  calcalc() カロリーを計算する
  [引数] pcal -- カロリーリストへのポインタ
         num  -- リストの要素数
  [戻り値] カロリー数
*******************************************/
float calcalc(CALORIE *pcal, int num)
{
    char name[40]; /* 入力した食品名 */
    float gram;    /* 入力したグラム数 */
    float totalcal = 0.0; /* 合計カロリー */
    int i;

    printf("--食品名一覧---------------¥n");
    for(i = 0; i < num; i++)
        printf("%s¥t", (pcal+i)->name);
    printf("¥n--------------------------¥n");

    while(1) {
        printf("食品名(endで計算) : ");
        scanf("%s", name);
        if(strcmp(name, "end") == 0)
            break;
        printf("グラム数 : ");
        scanf("%f", &gram);
        for(i = 0; i < num; i++) {
            if(strcmp(name, (pcal+i)->name) == 0) {
                totalcal += (pcal+i)->value * gram / 100.0;
                break;
            }
        }
    }
    return totalcal;
}
```

COLUMN

コラム

～プログラムの最適化～

プログラムが、その性能を最大限発揮できるように調整することを、最適化といいます。たとえば、大量の計算を実行するプログラムの場合、ひとつひとつの演算にかかる時間を、できるだけ短縮する必要があるでしょう。また、メモリが少ない環境で動くプログラムの場合、実行ファイルの大きさを、なるべく小さくする必要があるでしょう。

最も簡単な最適化の方法は、コンパイラオプションを設定することです。コンパイラオプションによって、プログラムの目的や動作環境に応じた、最適なコンパイル方法を指定することができます。たとえば、MicrosoftのCコンパイラには、次のようなコンパイラオプションが用意されています。

オプション	機能
/G5	Pentiumプロセッサに最適なコードを生成します
/Ot	処理の高速化を優先してコンパイルします
/Os	実行ファイルの最小化を優先してコンパイルします
・・・	・・・

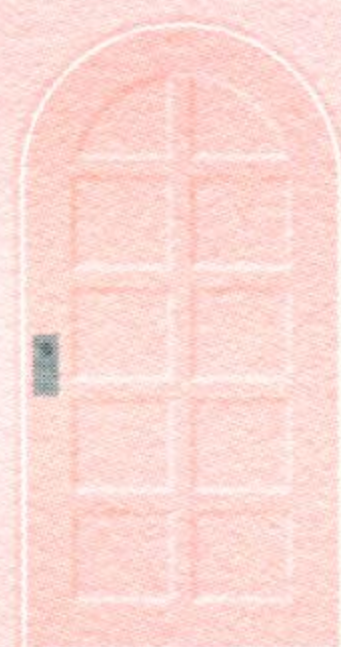
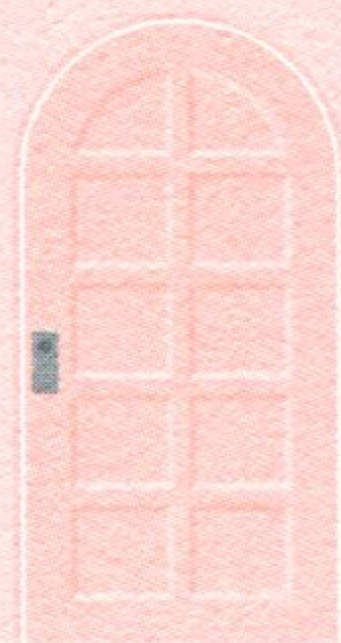
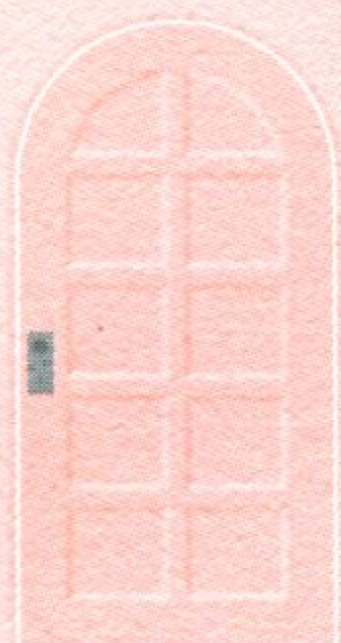
コンパイラ任せにするのではなく、プログラマ自身が最適化に気をくばり、ソースコードをチューニングすることも重要です。たとえば、この章で学んだ引数つきマクロと、関数にはそれぞれ次のような特徴があります。

引数つきマクロ
・処理がソースコードの中に埋め込まれるので、実行速度が速い
・使用頻度が多いと、ソースコードが大きくなる

関数
・呼び出しの度にジャンプする分、実行速度が遅くなる
・1箇所を参照するので、ソースコードは小さい

こうしてみると、一般論として、処理が比較的簡単で、呼び出しの回数が少ないようであれば、引数つきマクロにしたほうがよさそうです。一方、処理の内容が複雑だったり、呼び出しの回数が多い場合には、関数として実装したほうがよいと考えられます。この他にも、なるべくメモリを使わない変数の型を選択したり、よりよいアルゴリズム（プログラム構造）とデータ構造を検討したり、さまざまな点が、最適化のポイントになります。

ここ数年で、プロセッサの速度はどんどん速くなり、メモリやハードディスクの容量も飛躍的に増えました。実際のところ、昔ほど、最適化に気をくばる必要はなくなっています。しかし、いざ実用的なプログラムを組むことになったとき、最適化の問題はとても重要に感じられるはずです。まずは、無駄を省いたきれいなソースコードを書くところからはじめて、さまざまな最適化の手法を研究してみてください。

付　録

付　録

【高度なトピックス】

ここまで学習してきて、ちょっとしたプログラムを書くには十分な知識が身についているのではないかと思います。ここでは、「知っていると便利な情報」など、ハイレベルなプログラミングを目指すには欠かせない事柄を紹介しましょう。

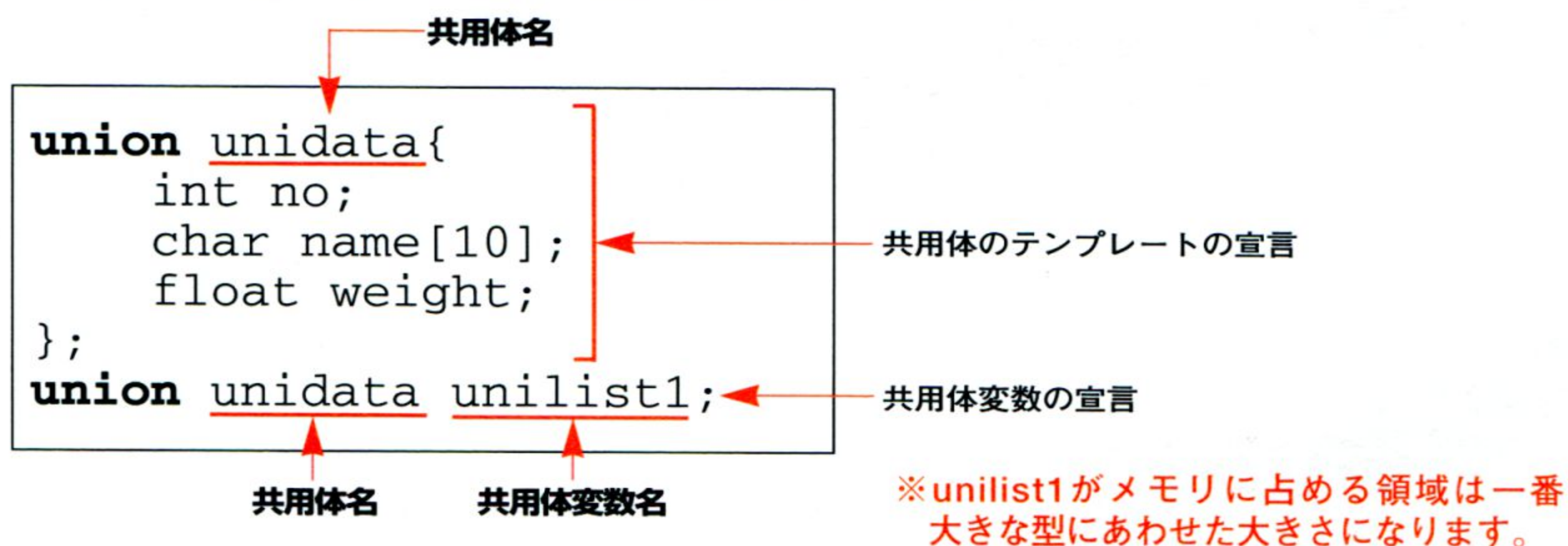
■共用体

1つのアドレスに異なるデータを割りあてる**共用体**という概念を紹介します。

≪共用体の概念≫

共用体を使うと、1つのメモリ領域で異なる型の変数のどれか1つを選んで使うことができます。共用体の宣言や参照の書式は構造体とほとんど同じです。

```
union unidata{
    int no;
    char name[10];
    float weight;
};
union unidata unilist1;
```


※unilist1がメモリに占める領域は一番大きな型にあわせた大きさになります。

≪共用体の使い方≫

構造体と同様に、共用体変数名とメンバ名をピリオドでつないで参照します。

```
unilist1.no = 1;
printf("%d¥n", unilist1.no);
strcpy(unilist1.name, "nagashima");
printf("%s¥n", unilist1.name);
unilist1.weight = 59.3;
printf("%f¥n", unilist1.weight);
```

メンバの値を活用するときは、必ず「最後に代入したメンバ」を参照してください。同じ記憶領域を共有しているので、でたらめな順番で参照しても値は保証されません。

```
unilist1.weight = 59.3;
unilist1.no = 1;
printf("%f¥n", unilist1.weight);
```

この時点でunilist1にはint型の1が入ります。

正しい値が表示されません。

例

```
#include <stdio.h>
union _user{
    int userid; /*ユーザID*/
    char name[10]; /*名前*/
} user;

main()
{
    int flag = 0;
    printf("入力項目は？(0=ID 1=名前) ");
    scanf("%d", &flag);
    if(flag) {
        printf("name? ");
        scanf("%s", user.name);
        printf("名前は%sですね。¥n", user.name);
    } else {
        printf("ID? ");
        scanf("%d", &(user.userid));
        printf("IDは%dですね。¥n", user.userid);
    }
}
```

実行結果

```
入力項目は？(0=ID 1=名前) 0 ↵
ID? 456 ↵
IDは456ですね。
```

※太字はキーボードから入力した文字

■列挙型

整数値に特定の名前を与える**列挙型**という概念を紹介します。列挙型を使うと、無味乾燥なプログラムを少しだけ見やすくできます。

≪列挙型の概念と宣言≫

列挙型を使うとint型の整数値に名前をつけることができます。

列挙型の宣言は**enum**（イーナム）という記述ではじめます。

次の例では、monthはJanuary～Decemberのいずれかの値を取ります。

列挙型名

```
enum _month{
    January,
    February,
    March,
        :
    November,
    December
} month;
```

列挙定数

列挙型変数名

それぞれの列挙定数は、0から順に1ずつ増やした整数値になります。

```
January    ・・・ 0
February   ・・・ 1
March      ・・・ 2
    :
November   ・・・10
December   ・・・11
```

```
enum _week{
    Sunday = 10,
    Monday,
    Tuesday = 15,
    Wednesday
        :
    Saturday,
} week;
```

任意の数を与えると、そこから1つずつ増えていきます。

```
Sunday     ・・・10
Monday     ・・・11
Tuesday    ・・・15
Wednesday  ・・・16
    :
Saturday   ・・・19
```

≪列挙型の活用≫

列挙型変数は列挙指定子のいずれかの名前を使った代入や参照ができます。

```
month = March;
```

※任意の値を代入できます。

■ビットやバイトに関する演算子

≪ビット演算子≫

コンピュータ内の情報をビット単位で比較、操作するときに使うのが次の**ビット演算子**です。

◎ 論理積（and）&

各ビットを比べて、「両方とも1なら1を、そうでなければ0」を返す演算です。

例）a = 170、b = 245のとき

変数名	10進数	2進数								
		b8	b7	b6	b5	b4	b3	b2	b1	
a	170	1	0	1	0	1	0	1	0	
		⇕	⇕	⇕	⇕	⇕	⇕	⇕	⇕	比較
b	245	1	1	1	1	0	1	0	1	
		↓	↓	↓	↓	↓	↓	↓	↓	両方とも1なら1
a & b	160	1	0	1	0	0	0	0	0	

◎ 論理和（or）|

各ビットを比べて、「片方でも1なら1を、そうでなければ0」を返す演算です。

例）a = 170、b = 245のとき

変数名	10進数	2進数								
		b8	b7	b6	b5	b4	b3	b2	b1	
a	170	1	0	1	0	1	0	1	0	
		⇕	⇕	⇕	⇕	⇕	⇕	⇕	⇕	比較
b	245	1	1	1	1	0	1	0	1	
		↓	↓	↓	↓	↓	↓	↓	↓	片方でも1なら1
a | b	255	1	1	1	1	1	1	1	1	

◎ 排他的論理和（xor）^

各ビットを比べて、「片方が1でもう片方が0なら1を、そうでなければ0」を返す演算です。

例）a = 170、b = 245のとき

変数名	10進数	2進数							
		b8	b7	b6	b5	b4	b3	b2	b1
a	170	1	0	1	0	1	0	1	0
		⇕	⇕	⇕	⇕	⇕	⇕	⇕	⇕ 比較
b	245	1	1	1	1	0	1	0	1
		↓	↓	↓	↓	↓	↓	↓	↓ 値が異なれば1
a ^ b	95	0	1	0	1	1	1	1	1

◎ 1の補数表現（not）~

各ビットを「反転させたもの」を返す演算です。

例）a = 170のとき

変数名	10進数	2進数							
		b8	b7	b6	b5	b4	b3	b2	b1
a	170	1	0	1	0	1	0	1	0
		↓	↓	↓	↓	↓	↓	↓	↓ 反転
~a	85	0	1	0	1	0	1	0	1

以上の演算をまとめると次のようになります。

Aのビット	Bのビット	A&B	A|B	A^B	~A
1	1	1	1	0	0
1	0	0	1	1	0
0	1	0	1	1	1
0	0	0	0	0	1

≪シフト演算子≫

ビット列を左右に指定した分だけずらす（シフトする）演算子を**シフト演算子**といいます。シフト演算子は次の2種類です。

◎右シフト演算子 >>

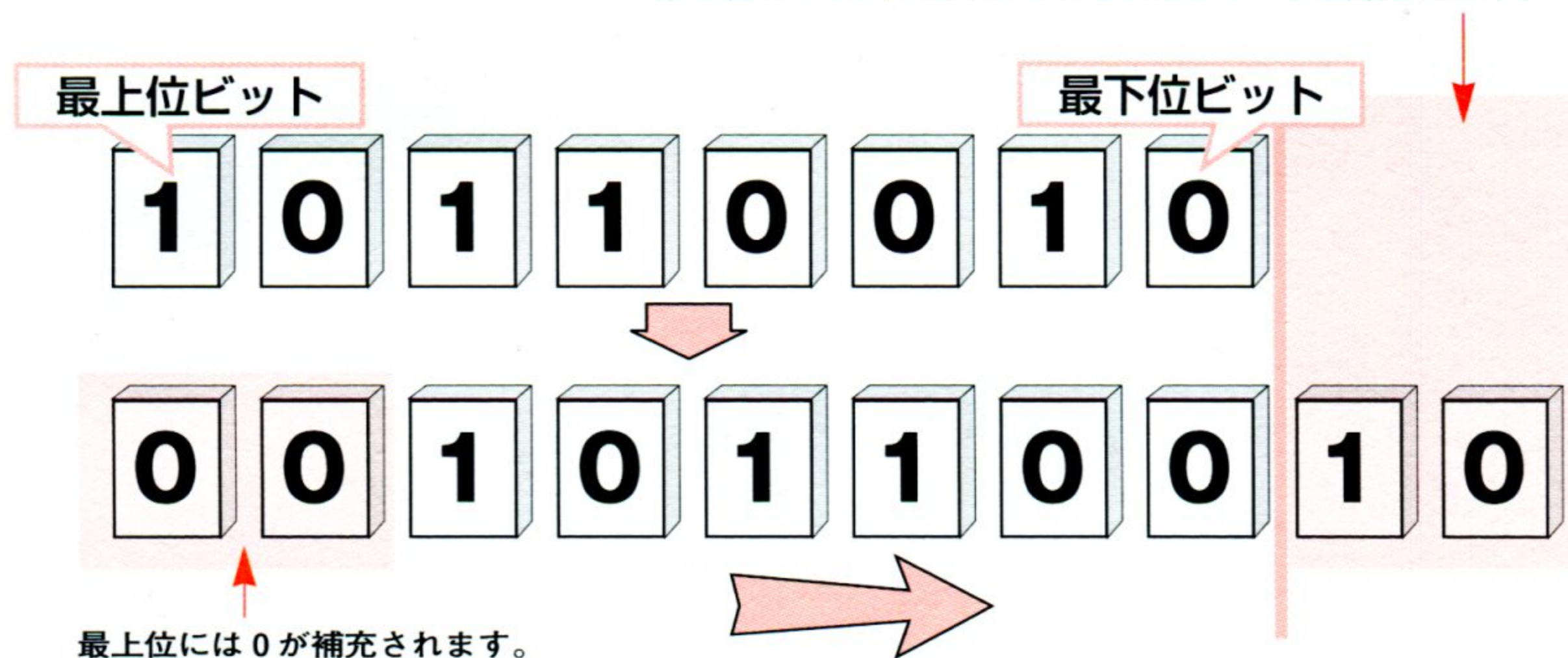
例）a >> 2 …右に2ビットシフトする

最下位ビットより右にシフトされたビットは破棄されます。

最上位ビット　最下位ビット

1 0 1 1 0 0 1 0

0 0 1 0 1 1 0 0 1 0

最上位には0が補充されます。

◎左シフト演算子 <<

例）a << 2 …左に2ビットシフトする

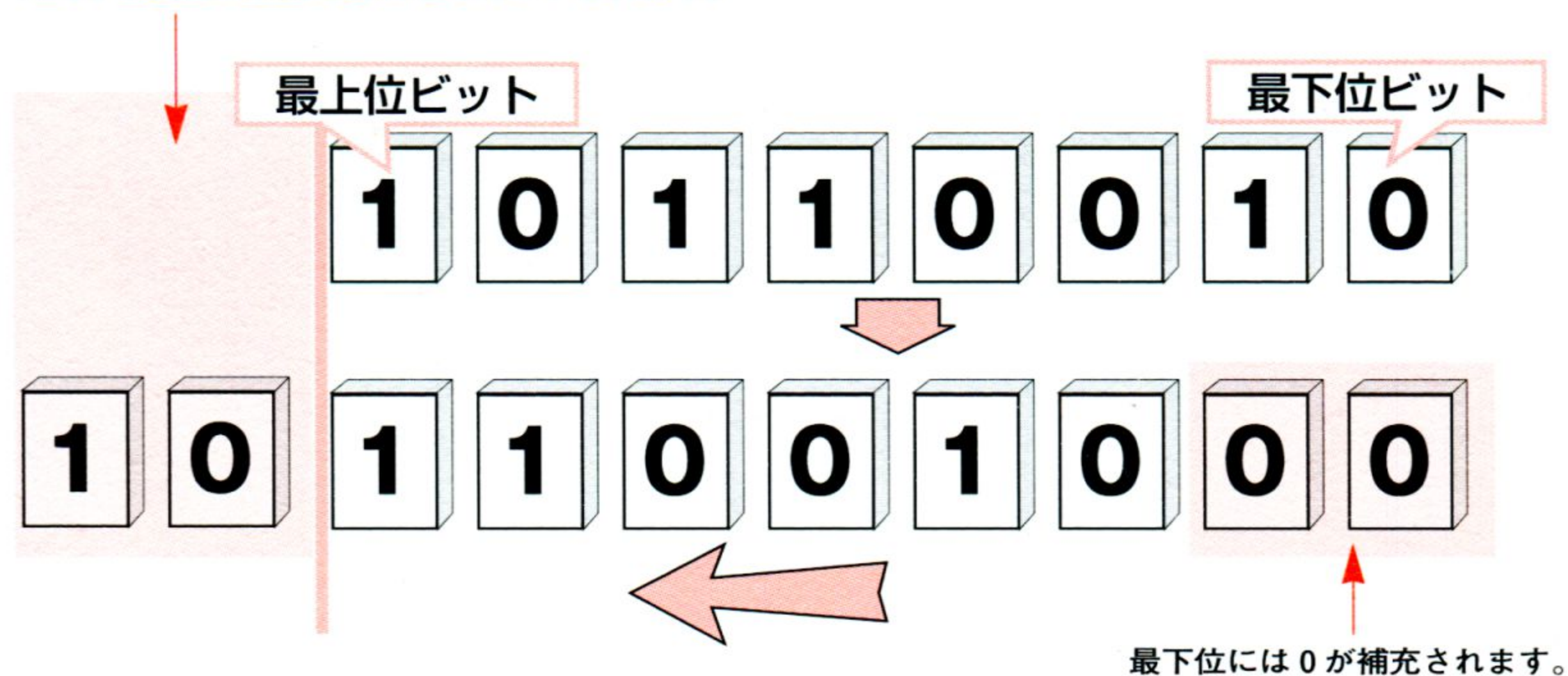
代入する変数の上限に合わせてカットされます。

最上位ビット　最下位ビット

1 0 1 1 0 0 1 0

1 0 1 1 0 0 1 0 0 0

最下位には0が補充されます。

例

```
#include <stdio.h>

main()
{
    char a = 10; ← 00001010
    char b = 9;  ← 00001001
    char c = 1; ←———— シフト量
    printf("%d & %d = %d¥n", a, b, a & b);
    printf("%d | %d = %d¥n", a, b, a | b);
    printf("%d ^ %d = %d¥n", a, b, a ^ b);
    printf("%d << %d = %d¥n", a, c, a << 1);
    printf("%d >> %d = %d¥n", a, c, a >> 1);
    printf("~%d = %d¥n", a, ~a);
}
```

実行結果

```
10 & 9 = 8       ←00001000
10 | 9 = 11      ←00001011
10 ^ 9 = 3       ←00000011
10 << 1 = 20     ←00010100
10 >> 1 = 5      ←00000101
~10 = -11        ←11110101
```

符号付整数型の内部では、最上位ビットを符号として使っています。
0は+を表し、残りのビットで数を表します。
1は-を表し、残りのビットから1を引いて反転したものが絶対値になります。

■時間に関する関数

現在の日時を取得する方法を紹介します。**time()**関数と**localtime()**関数を組み合わせて使えば、「○年×月△日※曜日、○時×分△秒」まで正確な時間を得ることができます。これらの定義や関数は**time.h**の中で定義されています。

≪現在時刻を得る≫

現在の時刻を得るには、time()関数を使います。なお、時間関係の関数や定義を使うには、time.hをインクルードする必要があります。

```
time_t ct;
ct = time(NULL);
```

time()関数で得られた値はtime_t型の変数に格納します。

経過時間の格納領域を指定しますが、通常はNULLでかまいません。

グリニッジ標準時で、1970/1/1 00:00:00より現在までの経過時間を秒数で返します。

※time_t型はtime.hで定義されています。

time()関数で得られた値は秒数で、しかも時差が考慮されていないので、そのまま使うのは困難です。そこで、localtime()関数を使ってこれを変換します。

```
struct tm *now;
now = localtime(&ct);
```

time()関数で得た値が入った変数のアドレスを指定します。

localtime()関数で得られた現在時刻の情報は、time.hですでに用意しているtm構造体変数に入ります。localtime()関数ではそのポインタのみを返します。

tm構造体のメンバは次のようなものです。

メンバ名	内容
tm_sec	秒(0～59)
tm_min	分(0～59)
tm_hour	時間(0～24)
tm_mday	日付(0～31)
tm_mon	月(0～11、1月を0とする)
tm_year	現在の西暦から1900を引いた値
tm_wday	曜日(0～6、日曜日を0とする)
tm_yday	年初からの通算日数(0～365、1月1日を0とする)
tm_isdst	夏時間が有効な場合には 0 以外の正の値、夏時間が無効な場合には0、夏時間かどうか不明な場合には負の値。C ランタイム ライブラリは、DST (Daylight Saving Time) の計算では合衆国の法律に従うことを前提としています。

≪時間に関するその他の関数≫

主な時間関係の関数を紹介します。

関数名	働き	使い方
localtime()	time_t型変数→tm構造体	time_t t; struct tm *ptmtime = localtime(&t);
gmtime()	time_t型変数→tm構造体 (時差を考慮しない)	time_t t; struct tm *ptmtime = gmtime(&t);
mktime()	tm構造体→time_t	struct tm tmtime; time_t t = mktime(&tmtime);
asctime()	tm構造体→文字列※	struct tm tmtime; char *s = asctime(&tmtime);
ctime()	time_t型変数→文字列※	time_t t; char *s = ctime(&t);

※ asctime()、ctime()で得られる文字列の形式は固定です。

例

```
#include <stdio.h>
#include <time.h>

main()
{
    time_t ct;
    struct tm *now;
    ct = time(NULL);
    now = localtime(&ct);

    printf("%d年%d月%d日%2d:%2d:%2d¥n",
        (now->tm_year)+1900,
        (now->tm_mon)+1, now->tm_mday,
        now->tm_hour, now->tm_min, now->tm_sec);
    printf("%s", ctime(&ct));
}
```

実行結果

```
2001年12月21日18:28:01
Sun Dec 21 18:28:01 2001
```

← 実行した瞬間の時間を表示します。

■数学関数

数学的な計算を行う関数を紹介します。算数レベルの計算は第2章で学んだ演算子で十分ですが、指数や平方根などの数学レベルの計算には、**math.h**で定義された数学用の関数を使います。

≪数学処理を行う関数≫

数学関連の主な関数を紹介します。以下の関数を使うにはmath.hをインクルードします。

関数名	働き	使い方	意味(mはint型、x、yはdouble型)
abs()	絶対値(整数)	int n = abs(m);	$n = \|m\|$
fabs()	絶対値(実数)	double a = fabs(x);	$a = \|x\|$
sqrt()	平方根	double a = sqrt(x);	$a = \sqrt{x}$
exp()	e指数	double a = exp(x);	$a = e^x$
log()	自然対数	double a = log(x)	$a = \log x$
pow()	べき乗	double a = pow(x,y);	$a = x^y$
log10()	常用対数	double a = log10(x);	$a = \log_{10} x$
sin()	サイン	double a = sin(x);	$a = \sin x$
cos()	コサイン	double a = cos(x);	$a = \cos x$
tan()	タンジェント	double a = tan(x);	$a = \tan x$

sin()、cos()、tan()の三角関数では角度をラジアンで指定します。
たとえば、20°のラジアン値は次のように求められます。

```
20.0*3.14/180.0
```

← ラジアン＝$\frac{角度[°] \times \pi}{180}$

※3.14159･･･というように細かく記述すれば、より正確な値を算出できます。

≪乱数を作る≫

乱数とは規則性のない数字のことです。プログラムで乱数を作るには**rand()**（ランダム）関数と**srand()**（エスランダム）関数を使います。これらの関数を使うにはstdlib.hをインクルードします。

プログラムで乱数を発生するには次のようにします。

```
int n;
srand(time(NULL));
n = rand();
```

srand( )関数で乱数発生の基準となる数字（seed）を指定します。

rand( )関数はseed値をもとに、int型の乱数を発生します。

※seed値が一定だと、プログラムの起動ごとに毎回同じ乱数ができてしまうので、現在時刻を利用するとよいでしょう。

例

```
#include <stdio.h>
#include <math.h>
#define PI 3.14159

main()
{
    int kakudo = 30;
    double a, b, c;
    a = sin(kakudo*PI/180);
    b = cos(kakudo*PI/180);
    c = tan(kakudo*PI/180);
    printf("角度%d度¥nsin %f¥ncos %f¥ntan %f¥n",
          kakudo, a, b, c);
}
```

math.hが必要です。

円周率をPIという名前で定義します。

実行結果

```
角度30度
sin 0.500000
cos 0.866026
tan 0.577350
```

■サーチとソート

C言語には、配列の並べ替えとデータ検索を高速に行うための関数が備わっています。

≪データを並べ替える≫

配列内の数値や文字列を並べ替えるときは**qsort()**（クイックソート）関数を使います。qsort()関数を使うにはヘッダファイルstdlib.hをインクルードします。

qsort()関数は次のように記述します。

```
int nums[] = {4, 6, 1, 3 };
qsort(nums,4,sizeof(int),compare);
```


比較関数は、1回分の比較処理を定めた関数で、次のような形式で自分で定義する必要があります（関数名はcompareでなくてもかまいません）。a、bには比較される各要素が入ります。

```
int compare(const void *a, const void *b)
{
    *a > *b のとき ・・・正の値を返す
    *b > *a のとき ・・・負の値を返す
    *a = *b のとき ・・・0を返す
}
```

※これは昇順の例です。降順のときは戻り値の符号を逆にします。

≪データを検索する≫

配列（ただしソートされていること）から指定したデータを探すには**bsearch()**（バイナリサーチ）関数を使います。データが見つかった場合は、その配列要素へのポインタを返し、見つからない場合はNULL値を返します。bsearch()関数を使うには、ヘッダファイルstdlib.hをインクルードします。

```
const int a = 1;
bsearch(&a, nums, 4, sizeof(int), compare);
```

例

```
#include <stdio.h>
#include <stdlib.h>

int compare(const void *a, const void *b)
{
    int x = *((int *)a);
    int y = *((int *)b);
    if(x > y)
        return 1;
    else if(x < y)
        return -1;
    else
        return 0;
}

main()
{
    int nums[10] = {4, 8, 3, 7, 5, 2, 9, 1, 6, 10};
    int a = 7, i;
    int *p;

    qsort(nums, 10, sizeof(int), compare);
    for (i = 0; i < 10; i++)
        printf("%d ", nums[i]);

    printf("¥n%dを検索します¥n", a);
    p = (int *) bsearch(&a, nums, 10, sizeof(int), compare);
    if(p == NULL)
        printf("%dは見つかりませんでした¥n", a);
    else
        printf("%dは配列nums[%d]にあります¥n", a, p-nums);
}
```

void型のポインタをint型ポインタにキャストして、そこにある値を求めます。

昇順に並べかえるための比較関数を定義します。

昇順に並べかえを行います。

昇順に並んだデータに対し、検索を行います。

実行結果

```
1 2 3 4 5 6 7 8 9 10
7を検索します
7は配列nums[6]にあります
```

7が見つかったのでelse以下を処理します。

■プログラムを中止する

エラー発生など何らかの事情によりプログラムを途中で終了したい、そんなときには**exit()**(イグジット)関数を使います。exit()関数を使えば、任意の個所でプログラムを正常に終了することができます。また、プログラムはその時点で開いているファイルをすべて閉じ、確保しているメモリをすべて解放します。

exit()関数を使用するにはヘッダファイルstdlib.hをインクルードし、次のように記述します。

```
exit(0);
```

終了コード
プログラム終了時にシステムに返す値を指定します。
一般的には以下のように設定します。

正常終了・・・EXIT_SUCCESS または 0
異常終了・・・EXIT_FAILURE や0以外の値

exit()関数は、どこにでも、いくつでも記述できます。次のように、プログラムがすべて正常に動作してから終了する場合と、エラー発生によりプログラムを途中で終了する場合とで、引数を変えてexit()関数を呼び出せば、システム側でプログラムの動きを知ることができます。

```
if((fp = fopen(file1.txt, "r")) == NULL) {
    printf("ファイルが存在しません¥n");
    exit(EXIT_FAILURE);  ← file1.txtというファイルが存在しない場合にプログラムを終了します。
};
            :
exit(EXIT_SUCCESS);
```

■サンプルプログラム

≪数あてゲーム本格版≫

第6章で紹介した数あてゲームを、乱数を使って、ちゃんと遊べるようにしたものです。

ソースコード

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX_NUM 100

int main()
{
    int n = -1;
    int answer;

    /* 乱数の発生 */
    srand(time(NULL));
    answer = (rand() % MAX_NUM) + 1;

    printf("数あてクイズ！1～%dの数字を入力してね¥n", MAX_NUM);
    while(n != answer) {
        scanf("%d",&n);
        if(n == answer-1 || n == answer+1)
            printf("おしい！¥n");
        else if(n > answer+1)
            printf("もっと小さい数です¥n");
        else if(n < answer-1)
            printf("もっと大きい数です¥n");
    }
    printf("正解です！¥n");
    return 0;
}
```

int n = -1; ← 偶然、正解にならないように、ありえない数にしておく。

answer = (rand() % MAX_NUM) + 1; ← 余りを取ることで、数の範囲を制限する。

実行結果

※太字はキーボードから入力した文字

数あてクイズ！1～100の数字を入力してね
50↵
もっと大きい数です
65↵
もっと大きい数です
75↵
もっと大きい数です
80↵
もっと大きい数です
90↵
もっと小さい数です
85↵
おしい！
86↵
正解です！

≪テキストファイルの内容をソートする≫

ソースファイル自身を最下行から逆向きに表示する"revtxt"を作ります。1行文字数は改行含め255文字までですが、行数は必要なだけ動的に確保します。

ソースコード

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <malloc.h>

typedef struct _TXTLN {
    int number;
    char string[256];
} TXTLN;

int main()
{
    FILE *fp;
    int i, lines;
    char s[256];
    TXTLN *p;

    /* 行数をカウントして、必要なメモリを確保し、読み込む */
    if(!(fp = fopen("revtxt.c", "r")))
        return 1;
    for(lines = 0; !feof(fp); lines++)
        fgets(s, 255, fp);
    p = (TXTLN *)malloc(lines * sizeof(TXTLN));
    if(!p)
        return 2;
    memset(p, 0, lines * sizeof(TXTLN));
    fseek(fp, SEEK_SET, 0);
    for(i = 0; i < lines; i++) {
        p[i].number = i;
        fgets(p[i].string, 255, fp);
        p[i].string[strlen(p[i].string)-1] = '¥0';
    }
    fclose(fp);

    /* 反対側から表示 */
    for(i = lines-1; i >= 0; i--)
        printf("%04d: %s¥n", p[i].number+1, p[i].string);

    free(p);
    return 0;
}
```

- typedef struct部分 ← テキストファイルの1行分を表す構造体
メンバは元の行番号と文字列（1行256文字まで）。
- for(lines = 0; ...) ← for文の継続条件では、必ずしもカウンタを使わなくてもよい。
- memset(...) ← 0に初期化
- p[i].string[...] = '¥0'; ← fgets( )関数は改行も読み込むので、改行を削除しておく。

実行結果

```
0041:
0040: }
0039:     return 0;
0038:     free(p);
0037:
0036:             printf("%04d: %s¥n", p[i].number+1, p[i].string);
0035:     for(i=lines-1; i>=0; i--)
0034:     /* 反対側から表示 */
0033:
0032:     fclose(fp);
:
```

上のソースコードの最下行から順に表示されます。

【開発の実際】

■プログラミング手法

プログラミングを習得するには、実際に手を動かしてみるのが一番です。自分でプログラムを書き、実行させることができれば身につくのも早く、なにより楽しく勉強を進められるでしょう。そこで、ここでは実際にプログラミングを行うときの手助けになるような情報を提供します。

C言語でプログラムを作成し、コンパイルするには、Cコンパイラが必要です。最近ではコンパイラだけではなく、開発に必要なツールを集めた製品（開発環境）が数多く出ています。代表的なものを次にあげてみます。

◎ Microsoft Visual C++

Windows上で動作するMicrosoft社（http://www.asia.microsoft.com/japan/）製の開発ツールです。Windowsで動作するあらゆるタイプのアプリケーションを開発できます。事実上、Windows開発環境のスタンダードといってよいでしょう。なお、Visual Studioは、Visual C++を含む複数の開発ツールの総称です。

◎ Borland C++ Builder

Windows上で動作するBorland社（http://www.borland.co.jp/）製の開発ツールです。この製品のコンパイラ部分のみの製品、C++ Compilerについては、Webからダウンロードして無料で利用できます。

◎ LSI C-86

LSI Japan社（http://www.lsi-j.co.jp/）のCコンパイラです。フリーで利用できる「試食版」が配布されていて、MS-DOS上（ただしいわゆる64KBの壁がある）で利用することができます。

◎ gcc

フリーソフトウェアを開発しているGNUプロジェクトのCコンパイラです。LinuxなどのUNIX系OSにおけるC言語の開発ではgccを利用するのが主流になっています。なお、redhat社のcygwin（http://sources.redhat.com/cygwin/）をインストールすれば、Windows環境でも使用できます。

以降は、Windows、UNIXそれぞれの代表として、Visual C++とgccでの開発について見ていくことにします。

≪Visual C++の統合開発環境による開発≫

Visual C++は、コンパイラをはじめ、さまざまなツールを含む統合開発環境（IDE）で、プログラムの編集、コンパイル、デバッグ、実行ができます。ウィンドウやアイコンを利用したGUIアプリケーションも作成できますが、本書の趣旨である、GUIを使わないアプリケーションももちろん開発できます。その方法をVisual C++ 6.0を例に解説していきましょう。

◎ プロジェクトの作成

Visual C++でプログラムを作成するには、まず「プロジェクト」を用意します。1つのプログラムは、複数のソースファイルからできていることが多いのですが、これらのファイルを管理する単位が、プロジェクトです。

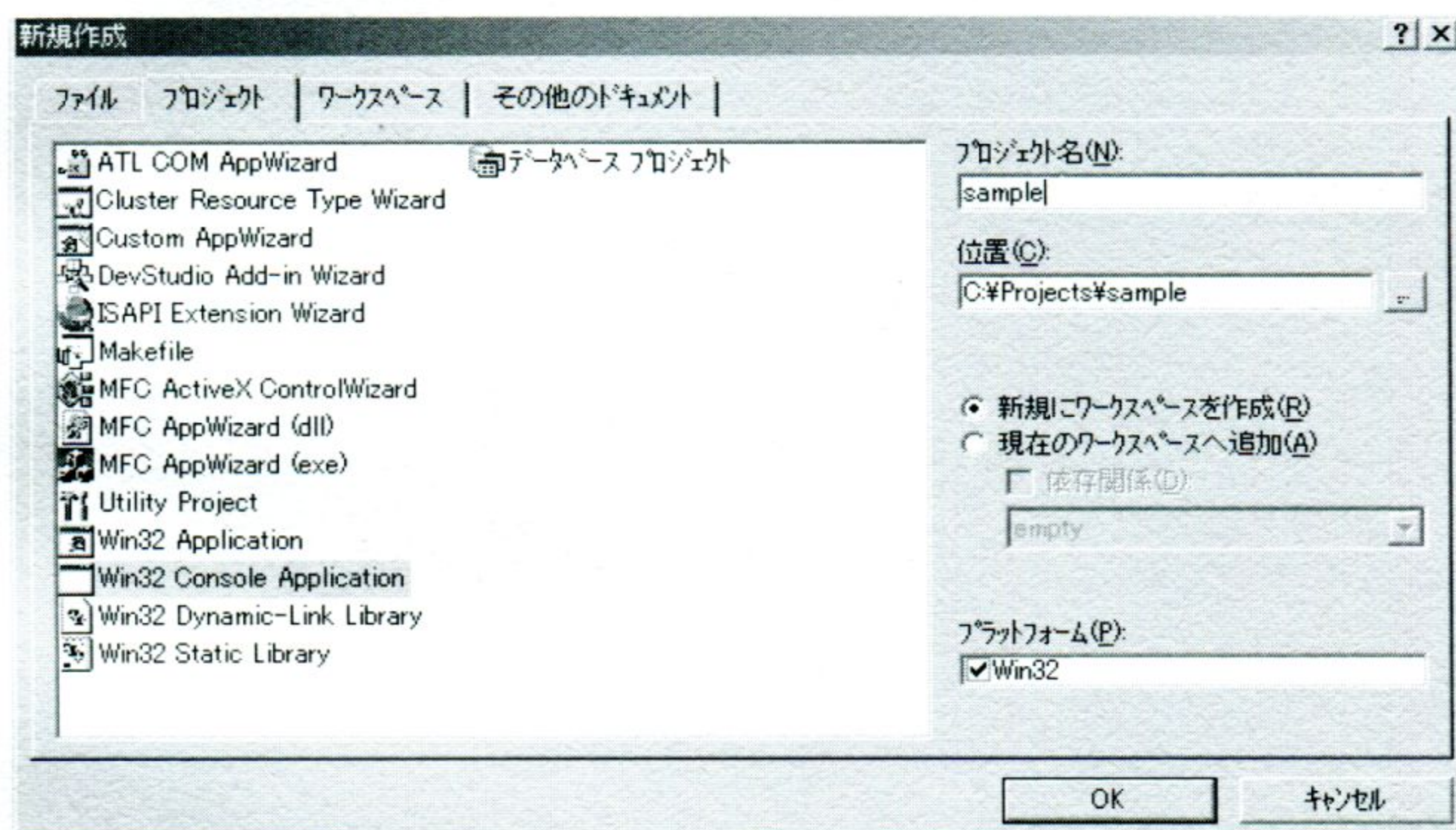



図1　プロジェクトの新規作成

Visual C++を起動したら、メニューの［ファイル］→［新規作成］を選び、新規作成ダイアログボックスの「プロジェクト」タブ（図1）で「Win32 Console Application」を選びます。「プロジェクト名」にはプロジェクトの名前（実行ファイルの名前になる）を指定します。必要であればプロジェクトを作成する「位置」を修正します。［OK］ボタンを押すと、さらにアプリケーションの種類を選択するダイアログボックスが現れますので、「空のプロジェクト」を選んで［終了］ボタンを押してください（「"Hello, World!" アプリケーション」を選んで、改造していくこともできますが、余計な機能がついてしまいます）。最後にプロジェクトの情報を確認して［OK］ボタンを押せば、指定されたフォルダにプロジェクトが作成されます。

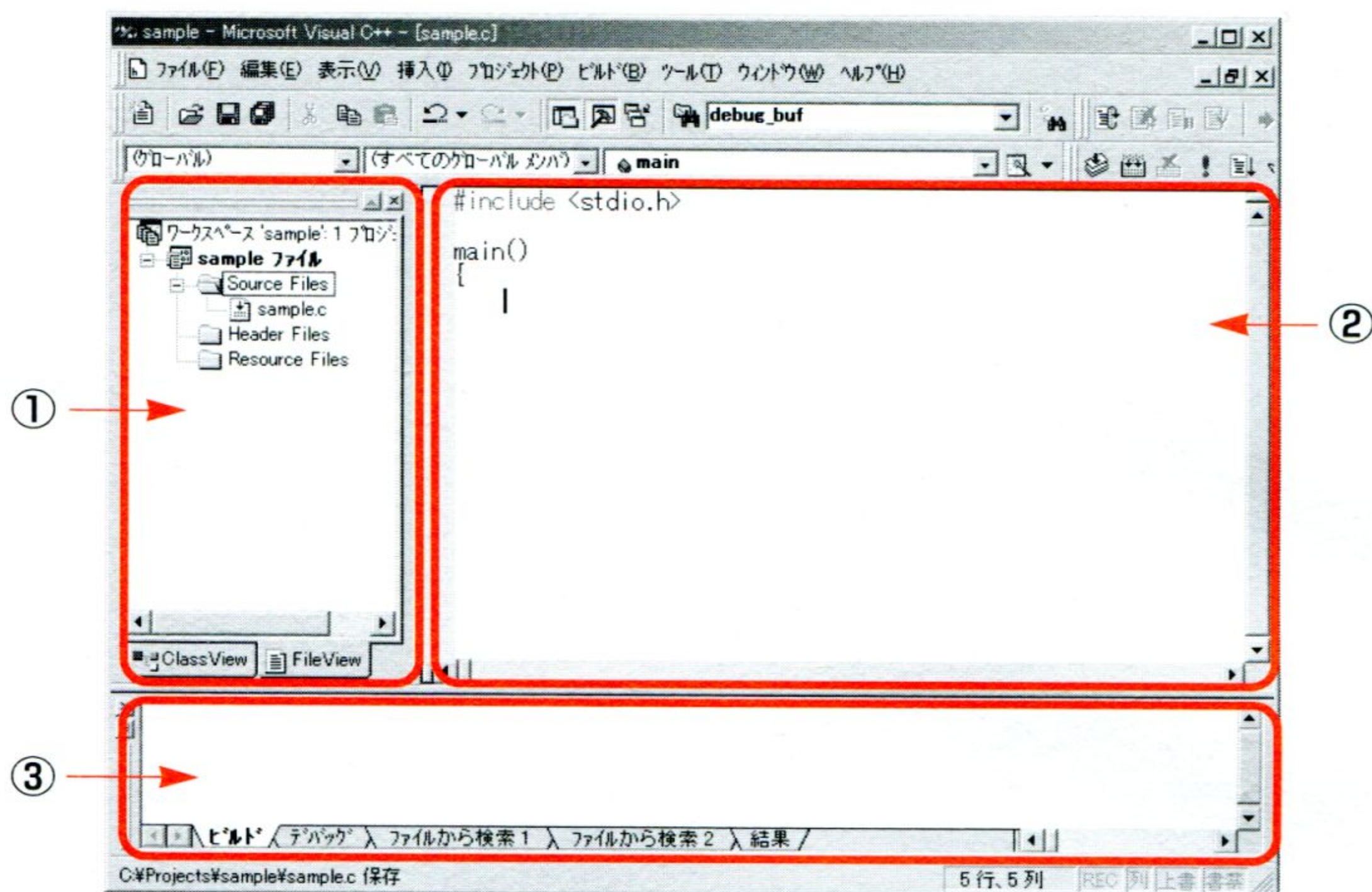



図2　ワークスペース

ところで、Visual C++では、作業の単位を「ワークスペース」と呼んでいます。1つのワークスペースでは複数のプロジェクトを扱うことができます。今回のようにプロジェクトを作成した直後は、ワークスペースが1つ、プロジェクトが1つ、ソースファイルは0という状態になっています。次回、プログラミングを再開する際は、メニューの［ファイル］→［ワークスペースを開く］で、目的のディレクトリのワークスペースファイル（*.dsw）を開きます（図2）。

◎ ソースファイルの追加

次のステップとして、プロジェクトにC言語のソースファイルを追加します。ソースファイルを新しく作るには、メニューの［ファイル］→［新規作成］を選び、新規作成ダイアログ（図3）の「ファイル」タブで「C++ ソース ファイル」を選択します。ファイル名（拡張子は.c）を入力し、「プロジェクトへ追加」がチェックされていることを確認して、［OK］ボタンを押すと、プロジェクトに新しいファイルが追加されます。プロジェクトに登録されているファイルは、ワークスペースウィンドウ（図2①）の「FileView」タブの中に表示されます。

すでにあるファイルをプロジェクトに追加したいときは、メニューから［プロジェクト］→［プロジェクトへ追加］→［ファイル］を選び、目的のファイルを指定してください。

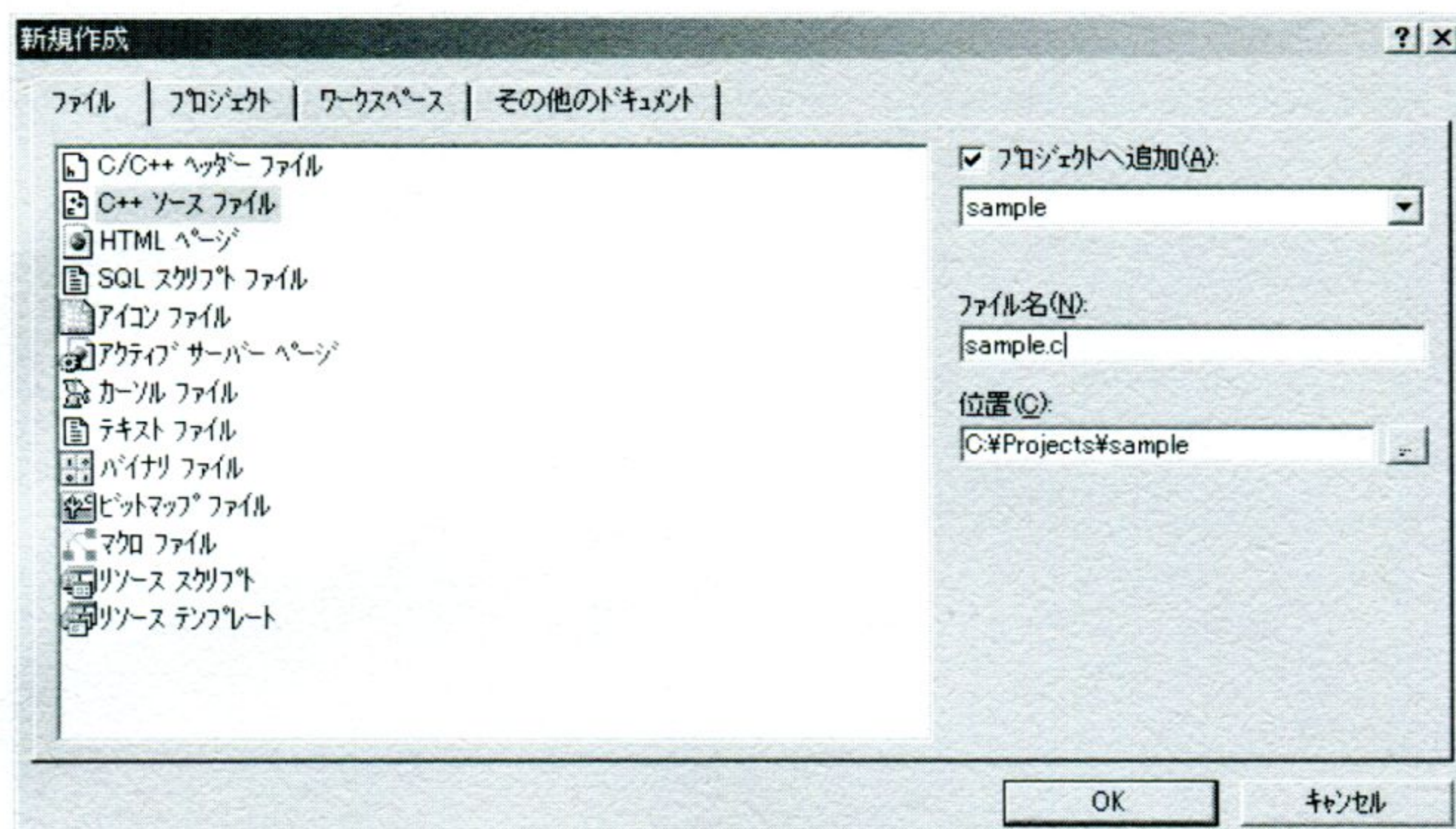



図3　ソースファイルを追加

◎ プログラムの編集・ビルド・実行

「FileView」タブからソースファイルを選ぶと、その中身が図2②の部分に表示されますので、プログラムコードを記述していきます。記述がおわったら、メニューの［ビルド］→［ビルド］を選んで、プログラムをビルドします。プログラムが正しければ、Visual C++はアウトプットウィンドウ（図2③）に「エラー0」と表示します。もしエラーがあれば、エラーの数と内容を表示します。また、エラーでなくても文法的に推奨されない記述があると、警告として報告します。

ビルドが成功したら、プログラムを実行できます。メニューから［ビルド］→［実行］を選ぶとコマンドプロンプトが開き、その中でプログラムが実行されます。プログラムが終了すると、「`Press any key to continue`」と表示されるので、何かキーを押すと、コマンドプロンプトウィンドウが閉じます。

なお、Visual C++は、先ほど指定したプロジェクトの「位置」のDebugディレクトリの下に、実行ファイルを作ります。たとえば、プロジェクトが「C:¥Projects¥sample」にあれば、「C:¥Projects¥sample¥Debug」に実行ファイル「sample.exe」を作ります。sample.exeを実行するには、Windowsのスタートメニューをたどって、「コマンドプロンプト」を開き、「sample.exe」を実行します（エクスプローラから直接実行しても構いませんが、結果を表示する間もなく、すぐに終了してしまいます）。なお、ファイルを扱うプログラムでは、実行するディレクトリの位置にも気をつけてください。

≪Visual C++のコマンドプロンプトでの開発≫

今までは、Visual C++のIDEを使ってプログラミングする方法を見てきましたが、IDEを使わずにプログラムをビルド・実行することもできます。そのときは、好みのテキストエディタでソースファイルを編集し、コマンドプロンプト内でコンパイルを行います。でも、その前に、少し準備をする必要があります。コマンドプロンプト内で、Visual C++をインストールしたフォルダの「bin」フォルダの中にある「VCVARS32.BAT」を実行してください。すると、コンパイルに必要な環境変数が設定され、コンパイラなどのツールが使えるようになります。

コマンドプロンプトでのコンパイルでは、「cl.exe」というプログラムを使います。このcl.exeこそコンパイラ本体です。sample.cというファイルをコンパイルするには、コマンドプロンプトでsample.cのあるフォルダに移動し、

```
cl sample.c
```

と入力すれば、同じフォルダに実行ファイルsample.exeが作成されます。

なお、プログラムを構成するファイルが複数ある場合は、それらのファイルをそれぞれコンパイルして、リンクしなければなりません。この手順を自動的に行うには、ファイル構成やそれらの順序を記述した「メイクファイル」が必要になります。メイクファイルの書き方については、かなり複雑なので割愛しますが、プロジェクトを作っていれば、メニューの［プロジェクト］→［メイクファイルのエクスポート］で、自動的に生成することもできます。Visual C++のメイクプログラムは、「nmake.exe」という名前です。「nmake <メイクファイル>」と入力すると、プログラムをビルドすることができます。

≪gccによるコンパイル≫

LinuxなどのUNIX系OSでは、Cプログラムのコンパイルにgccがよく使われます。gccはコンパイラプログラムの名前ですが、その開発環境全体を指すこともあります。

gccを使ったコンパイルの方法は、Visual C++でコマンドプロンプトを使う方法と同じような感じです。Emacsやviなどのエディタでソースファイルを編集し、完成したら、ソースファイル名を引数として、gccを起動します。ソースファイルが「sample.c」だとすると、次のように入力します。

```
gcc -o sample sample.c
```

-oオプションで指定しているのは、出力する実行ファイルの名前です。今回の例では、同じディレクトリにsampleという名前で実行ファイルが作成されます。なお、-oオプションを指定しないと、実行ファイル名はデフォルト名のa.outになります。

これを実行するときは、「./sample」と入力します。なお、gccで複数ファイルのコンパイルとリンクを行う場合にも、メイクファイルを利用できます（ただし、その書式はMS-DOSのときとは異なります）。

補足ですが、Cの標準ライブラリ関数を使う場合は、gccの-lオプションで必要なライブラリを指定します。-l<ライブラリ識別子名>とすると、標準ライブラリディレクトリの中の、lib+<ライブラリ識別子名>.aというライブラリファイルをリンクするようできます。

たとえば、sample.cでヘッダファイル「math.h」をインクルードしており、数学関数を使っている場合は、次のようにします。

```
gcc -o sample sample.c -lm
```

このようにすると、数学関数のライブラリlibm.aをリンクできます。どのようなライブラリが必要かは「man math」などと入力すれば、調べられます。

■デバッグ手法

プログラムにはバグ（間違い）がつきものです。どんなに優秀なプログラマでも、一度でバグのないプログラムを書くことはできません。プログラムが思ったように動かないときは、バグを発見し修正する、「デバッグ」という作業を行うことになります。

≪エラーの種類≫

プログラミングをしていて、最初に突きあたる壁がコンパイル時のエラーです。プログラムがコンパイルできない原因としては、文法が間違っている（シンタックスエラー）、コンパイルの方法が正しくないなどが考えられますが、どの部分が間違っているかはコンパイラが指摘してくれます。ただ、C言語のコンパイラが出力するエラーメッセージは、よくも悪くも「流れ作業的」で、1箇所間違いがあるだけでも、それ以降で整合性が取れないと、その都度メッセージを表示してしまいます。エラーメッセージがたくさん表示されたときは、あわてず、どのメッセージが本質的なものなのかを見極めて修正する必要があります。

さて、コンパイルできたからといって正しいプログラムができたというわけではありませ

ん。最も苦労するのはプログラム実行中の不具合です。バグといえば、通常はこちらの方を指します。バグには、プログラムが止まってしまう（ランタイムエラー）、止まらないがプログラムの動きがおかしい、動作は思ったとおりでも、間違った結果を出しているなどいろいろな種類があります。

たとえば、「i = 3;」は、変数iに値3を代入する文ですが、この文の等号（=）を間違えて2つ書いてしまったとすると、「i == 3;」となり、iと3が等しいかどうか比較する式になります。比較を行う式をそれだけで書いても意味はありませんが、文法的には正しいので、コンパイラはエラーを出しません。結局、意図していた「iに3を代入する」という処理が行われないプログラムができてしまいます。

≪バグの発見≫

プログラムのバグを発見するには、まずソースプログラムをじっくり読むことが基本です。自分の考えた処理が思ったとおりに記述されているかどうか、もう一度確認しましょう。しかし、それでもわからないときは、プログラムがどのように動作しているのかをよく調べる必要があります。

以降では、デバッグでよく使われるいくつかの手法について、その目的と方法を解説していきます。

◎処理を分割する

C言語では式や文の記述が非常に柔軟になっていて、工夫次第で凝ったものを作ることもできますが、バグの温床になることもあります。もし、まとめて書いていたら、処理や意味の単位である程度分けて書いてみましょう。ランタイムエラーのエラー情報には「○行目でエラーが起きました」という内容のものが多いので、1行にいくつもの文をまとめて書かない方がエラーの位置がわかりやすくなります。

演算子の優先順位も、慣れないと間違いやすいところです。式の意味がわかりにくい、優先順位がはっきりしないなどのときは、カッコをつけたり、一度変数に代入したりすると、意味がはっきりし、読みやすくなります。特にポインタや配列に関する演算、インクリメント演算などを多用すると複雑になるので、気をつけてください。

その昔、「困難は分割せよ」といった偉い人がいましたが、分割すれば、おのずと悪い部分が見えてくるものです。

◎**printf( )**を挿入する

コンパイルしたプログラムをただ実行したのでは、バグがあることはわかっても、原因ま

ではなかなかわかりません。そこで、ソースプログラムに、本来は必要ないprintf()を挿入して手がかりを得ます。たとえば、プログラム中に「printf("実行されました。¥n");」などと書いておけば、そこに到達したときにメッセージが出力され、その部分がいつ実行されるのかがわかるというわけです。さらに、変数の値を表示するようにしておけば、その時点での変数の値を調べることができます。

また、画面の出力内容やタイミングにシビアなアプリケーションでは、ログをファイルに残すのも有効な方法です。このときは、テキスト追加モード（"a"）でファイルをその都度オープンし、fprintf()でログを追加していきます。

なお、printf()やfprintf()で変数の内容を表示するときには、書式指定と変数の型の関係に気をつけましょう。int型の変数aに対し、「printf("%s", a);」などと書くと、新たなバグを生み出してしまいます。

◎関数ごとに実行する

C言語での処理の単位は関数なので、関数をテストすることは多いはずです。関数に様々な引数を与えて戻り値を調べれば、その関数が正常に動作しているかどうかがわかります。関数は様々な状況で呼び出されていますが、main()関数を書き換えて、目的の関数をすぐに実行できるようにすると効率的です。main()関数の一時的な書き換えが困難なときは、別のテスト用プログラムを作り、そこからテストしたい関数を呼び出すのもよいでしょう。

◎処理の流れを限定する

バグの潜んでいる場所を探すときは、条件分岐が邪魔になることがあります。条件分岐は状況に応じて動作が変わるので、バグがわかりにくくなるのです。このような場合は、条件分岐の条件式を書き換え、直接1（true）や0（false）を書いてしまうのも一つの方法です。この方法は、エラーチェックなどめったに実行されない部分をテストするためにも有効です。

◎データ構造を推測する

ときには、手を動かすだけではなく、推理力が必要になることもあります。複雑なアルゴリズムやデータ構造を持ったプログラムでは、バグの個所とはまったく関係のないところで異常な動作をすることもあるのです。そんなときは、ソースプログラムをいくら見返しても「こんなことが起こるわけがない」という結論にしかたどり着きません。

そこで、頭を働かせて、「この部分のメモリはどのように使われているだろう？」と考えてみます。自分自身で整理するために、データ構造を紙に書き出してみるのもいいでしょう。実際のところ、配列の範囲外の要素を参照したり、ポインタが間違ったところを指し

ていたりしてエラーになっているケースが多いのです。このようなバグを見つけるのは困難ですが、ある程度高度なプログラムの不具合の中では一番ありがちなものです。

≪デバッガの利用≫

ここまでは、ソースプログラムを変更してデバッグする方法を紹介してきました。これは、いわば自力でバグを見つけるための方法です。しかし、大きなプログラムになってくると、バグの発見やソースコードの書き換えが非常に大変になってきます。

そこで、ツールの力を借りることにします。デバッグを支援するツールのことを「デバッガ」といい、コンパイラと並んでプログラムの開発には欠かせないものです。たいていの開発環境はコンパイラとデバッガの両方を用意しています。代表的なデバッガの機能には、以下のようなものがあります。

◎ ブレークポイントの設定

デバッガを使えば、ソースプログラムの指定した位置でプログラムを一時停止することができます。ブレークポイントとは、その停止位置のことです。ブレークポイントを設定してプログラムを実行すると、その場所に来たときにプログラムが停止します。

◎ 変数の表示と変更

プログラムを一時停止したときに、そのときの変数の値を参照することができます。`printf()`の挿入と同じ機能ですが、こちらはソースプログラムを変更することなく、好きな変数の値を表示させることができます。また、多くのデバッガでは変数の値を直接書き換えて、その値のままでプログラムの実行を継続させることができるようになっています。

◎ ステップ実行

ソースプログラムの1行分ずつプログラムを実行する機能です。この機能を使えば、プログラムが実際にどのように動作しており、どの時点で止まってしまっているのかをとらえることができます。変数の値を表示させておけば、変数の値の移り変わりを調べることもできます。条件分岐が多いプログラムなどで特に便利な機能です。

このようにデバッガを使うと、ソースプログラムを変更するよりもはるかに簡単かつ効率的に、デバッグができます。デバッガが使える環境では、積極的にデバッガを利用して、労力と時間を節約しましょう。

≪Visual C++のデバッガ≫

具体的なデバッガとしてVisual C++のデバッガの利用方法を見てみましょう。Visual C++では、デバッガも同じ統合開発環境の中で利用できます。前のページで紹介したデバッグ手法に沿って、3つの基本的な機能の操作方法を紹介します（図4）。

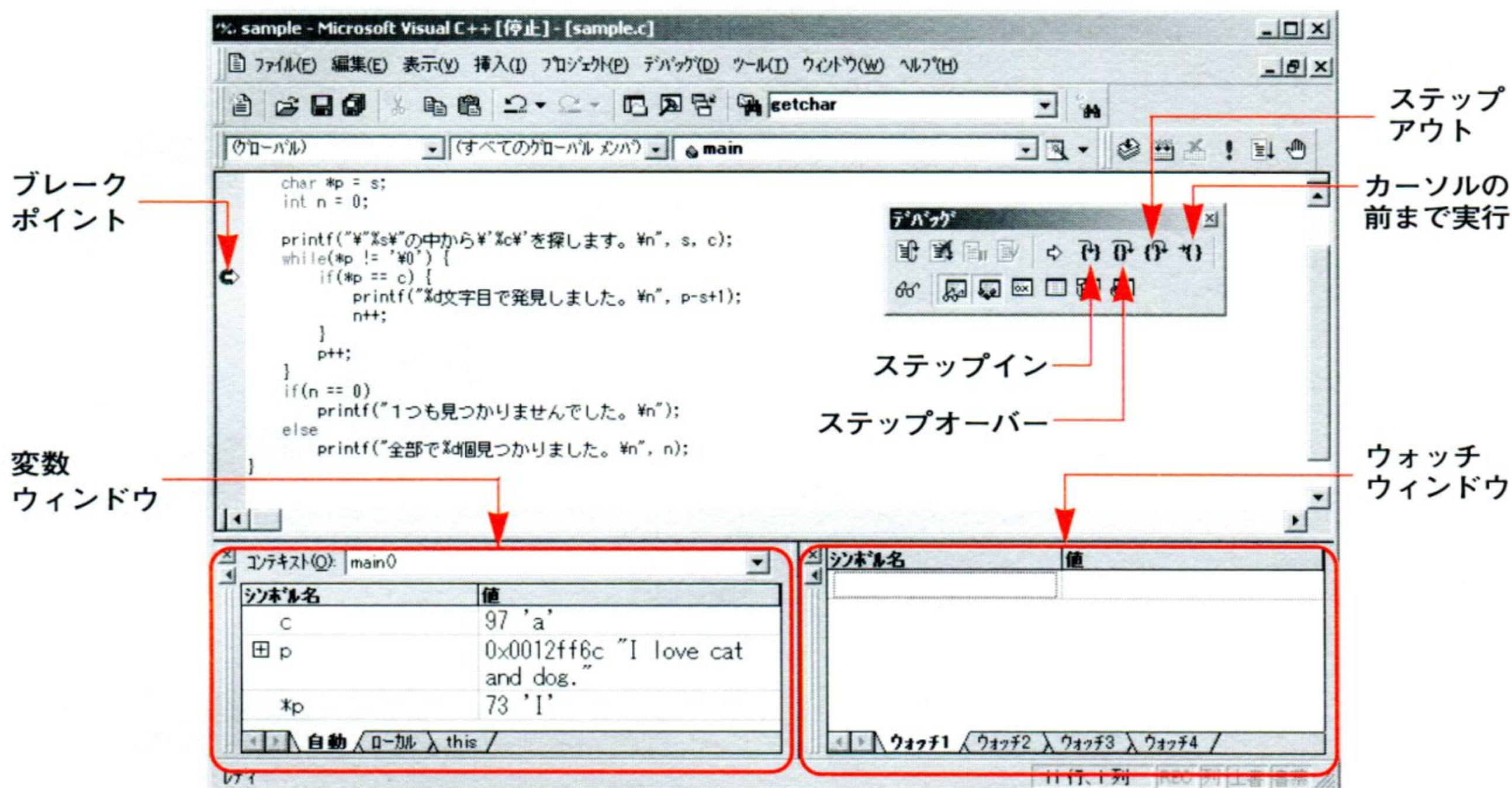



図4　Visual C++のデバッガ

◎ブレークポイントの設定

ソースプログラムの編集画面で直接ブレークポイントを設定することができます。ブレークポイントを設定するには、設定したい行にカーソルを移動し、[F9] キーを押します。もう一度 [F9] キーを押すと、ブレークポイントを解除できます。ブレークポイントはいくつでも設定できます。プログラムを実行するときは、通常の「実行（！マーク）」ではなく、デバッグ実行（[F5] キー）を選んでください。

◎変数の表示と変更

ブレークポイントでプログラムが停止すると、「変数」ウィンドウが開き、停止位置に関連のある変数とその値が表示されます。変数の値を継続的に表示させたいときは、「ウォッチ」ウィンドウにドラッグ&ドロップで追加することもできます。「ウォッチ」ウィンドウでは、変数の他にも自由に式を入力して、その値を表示させることができます。

◎ ステップ実行

Visual C++では、4種類のステップ実行ができます。

ステップオーバー（[F10] キー）はソースプログラムの1行分ずつプログラムを進めます。

ステップイン（[F11] キー）は、他の関数を呼び出しているとき、その関数の中に入っていきます。ステップアウト（[Shift] + [F11] キー）では現在実行中の関数を脱出します。

カーソルの前まで実行（[Ctrl] + [F10]）では、その名のとおり、カーソルのある位置でプログラムが止まります。ただし、その前にブレークポイントがある場合は、その位置で停止します。

これらの機能は、デバッグウィンドウの中のボタンを押すことでも実行できます。

【ASCIIコード表】

No	文字
0 (0x00)	NUL (null)
1 (0x01)	SOH (start of heading)
2 (0x02)	STX (start of text)
3 (0x03)	ETX (end of text)
4 (0x04)	EOT (end of transmission)
5 (0x05)	ENQ (enquiry)
6 (0x06)	ACK (acknowledge)
7 (0x07)	BEL (bell)
8 (0x08)	BS (backspace)
9 (0x09)	HT (horizontal tab)
10 (0x0A)	LF (line feed)
11 (0x0B)	VT (vertical tab)
12 (0x0C)	FF (form feed)
13 (0x0D)	CR (carriage return)
14 (0x0E)	SO (shift out)
15 (0x0F)	SI (shift in)
16 (0x10)	DLE (data link escape)
17 (0x11)	DC1 (device control 1)
18 (0x12)	DC2 (device control 2)
19 (0x13)	DC3 (device control 3)
20 (0x14)	DC4 (device control 4)
21 (0x15)	NAK (negative acknowledge)
22 (0x16)	SYN (synchronous idle)
23 (0x17)	ETB (end of trans. block)
24 (0x18)	CAN (cancel)
25 (0x19)	EM (end of medium)
26 (0x1A)	SUB (substitute)
27 (0x1B)	ESC (escape)
28 (0x1C)	FS (file separator)
29 (0x1D)	GS (group separator)
30 (0x1E)	RS (record separator)
31 (0x1F)	US (unit separator)

No	文字
32 (0x20)	(スペース)
33 (0x21)	!
34 (0x22)	"
35 (0x23)	#
36 (0x24)	$
37 (0x25)	%
38 (0x26)	&
39 (0x27)	'
40 (0x28)	(
41 (0x29)	)
42 (0x2A)	*
43 (0x2B)	+
44 (0x2C)	,
45 (0x2D)	-
46 (0x2E)	.
47 (0x2F)	/
48 (0x30)	0
49 (0x31)	1
50 (0x32)	2
51 (0x33)	3
52 (0x34)	4
53 (0x35)	5
54 (0x36)	6
55 (0x37)	7
56 (0x38)	8
57 (0x39)	9
58 (0x3A)	:
59 (0x3B)	;
60 (0x3C)	<
61 (0x3D)	=
62 (0x3E)	>
63 (0x3F)	?

No	文字
64 (0x40)	@
65 (0x41)	A
66 (0x42)	B
67 (0x43)	C
68 (0x44)	D
69 (0x45)	E
70 (0x46)	F
71 (0x47)	G
72 (0x48)	H
73 (0x49)	I
74 (0x4A)	J
75 (0x4B)	K
76 (0x4C)	L
77 (0x4D)	M
78 (0x4E)	N
79 (0x4F)	O
80 (0x50)	P
81 (0x51)	Q
82 (0x52)	R
83 (0x53)	S
84 (0x54)	T
85 (0x55)	U
86 (0x56)	V
87 (0x57)	W
88 (0x58)	X
89 (0x59)	Y
90 (0x5A)	Z
91 (0x5B)	[
92 (0x5C)	\
93 (0x5D)	]
94 (0x5E)	^
95 (0x5F)	_

No	文字
96 (0x60)	`
97 (0x61)	a
98 (0x62)	b
99 (0x63)	c
100 (0x64)	d
101 (0x65)	e
102 (0x66)	f
103 (0x67)	g
104 (0x68)	h
105 (0x69)	i
106 (0x6A)	j
107 (0x6B)	k
108 (0x6C)	l
109 (0x6D)	m
110 (0x6E)	n
111 (0x6F)	o
112 (0x70)	p
113 (0x71)	q
114 (0x72)	r
115 (0x73)	s
116 (0x74)	t
117 (0x75)	u
118 (0x76)	v
119 (0x77)	w
120 (0x78)	x
121 (0x79)	y
122 (0x7A)	z
123 (0x7B)	{
124 (0x7C)	|
125 (0x7D)	}
126 (0x7E)	~
127 (0x7F)	DEL (delete)

さくいん

＜記号・数字＞

＜A＞

＜B＞

＜C＞

[著者紹介]

(株)アンク(http://www.ank.co.jp/)

ソフトウェア開発から、Webサイト構築・デザイン、書籍執筆まで幅広く手がける会社。著書にホームページ辞典、HTMLタグ辞典など多数。

執筆　小林 麻衣子、宮地 弘子、高橋 誠
執筆協力　村上 拓真、相澤 奈美子、福田 有佐、福田 輝和
イラスト　小林 麻衣子

装丁・本文デザイン　嶋 健夫
編集　小川 史晃

Cの絵本 C言語が好きになる9つの扉

2002年 3月15日　初版第 1刷発行
2009年 5月10日　初版第22刷発行

著　者　(株)アンク(かぶしきがいしゃあんく)
発行人　佐々木 幹夫
発行所　株式会社 翔泳社(http://www.shoeisha.co.jp/)
印刷　昭和情報プロセス株式会社
製本　株式会社 国宝社

本書へのお問合せについては、iiページに記載の内容をお読みください。

落丁・乱丁はお取り替えいたします。03-5362-3705までご連絡ください。

ISBN4-7981-0103-6　Printed in Japan

ISBN4-7981-0103-6
C3055 ¥1380E
9784798101033
株式会社 翔泳社
定価：本体 1,380円十税
1923055013803

Cの絵本

Cの絵本

C言語が好きになる9つの扉

㈱アンク 著

SE
SHOEISHA